conference

proceedings

Java™ Virtual Machine Research and Technology Symposium (JVM '01)

Monterey, California USA
April 23–24, 2001

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX
Java™ Virtual Machine Research and Technology Symposium
Monterey, California, April 2001

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $24 for members and $30 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

ISBN 1-880446-11-1

Printed in the United States of America on 50% recycled paper, 10–15% post-consumer waste.

USENIX Association

Proceedings of the Java™ Virtual Machine Research and Technology Syposium (JVM '01)

April 23–24, 2001
Monterey, California, USA

Symposium Organizers

Program Chair

Saul Wold, *Sun Microsystems, Inc.*

Program Committee

Tony Cocchi, *IBM*
Urs Hoelzle, *University of California, Santa Barbara*
Juergen Kreileder, *The Blackdown Project*
Tim Lindholm, *Sun Microsystems*
Matt Welsh, *University of California, Berkeley*

The USENIX Association Staff

Java™ Virtual Machine Research and Technology Symposium

April 23–24, 2001

Monterey, California, USA

JVM Potpourri
Session Chair: Saul Wold, Sun Microsystems, Inc.

Garbage Collection
Session Chair: Juergen Kreileder, Blackdown

Small Devices
Session Chair: Tim Lindholm, Sun Microsystems, Inc.

Message from the Symposium Chair

Over the past few years the Java language has experienced enormous growth. Much work has also been going on under the covers in code generators, garbage collection, threading improvements, and other performance and research areas. Historically this work has been presented here and there, scattered over various conferences and symposia. By bringing it together in one place we can develop a better overall picture of what is happening in the academic and corporate research arenas.

Welcome, therefore, to the Java Virtual Machine Research and Technology Symposium.

We've created something new: a small but powerful symposium. We solicited around the globe for papers. From the 50 papers received, we were able to select 18 for presentation. The accepted papers come from all regions of the world, including Japan, Europe, and the Americas. We have a good mix of corporate and academic works. We are also going to have an exciting collection of Work-in-Progress presentations.

The program is the result of the hard work of many. First, we want to thank the authors who submitted papers, including those whose papers we weren't able to accept. Next, we want to thank the program committee members for their diligent labor. They read a large number of papers in a short time. They also shepherded the accepted papers. We also want to thank the external reviewers, who contributed numerous high-quality reviews. Finally, we would like to thank the USENIX staff for all of their detailed and timely help in putting the conference together.

Our thanks to each of these individuals for their contributions towards producing the outstanding JVM '01 program you see before you. It's been our privilege working with each of you.

Saul Wold
Program Chair

Proceedings of the

Java™ Virtual Machine

Research and Technology Syposium

(JVM '01)

The Java HotSpot™ Server Compiler

Michael Paleczny
Christopher Vick
Cliff Click

Sun Microsystems
901 San Antonio Road, Palo Alto, CA 94303
{michael.paleczny,christopher.vick,cliff.click}@eng.sun.com

Abstract

The Java HotSpot™ Server Compiler achieves improved asymptotic performance through a combination of object-oriented and classical-compiler optimizations. Aggressive inlining using class-hierarchy analysis reduces function call overhead and provides opportunities for many compiler optimizations.

1. Introduction

Performance for the Java™ Platform has evolved in stages. Early VM's were interpreter-only. Later VM's were interpreter plus template generated code, and finally interpreter plus optimized code. The Java HotSpot Virtual Machine™ improves performance through optimization of frequently executed application code. The Client version provides very fast compilation times and a small footprint with modest levels of optimization. The Server version applies more aggressive optimizations to achieve improved asymptotic performance. These optimizations include class-hierarchy aware inlining, fast-path/slow-path idioms, global value-numbering, optimistic constant propagation, optimal instruction selection, graph-coloring register allocation, and peephole optimization.

The first section describes the runtime environment that both the compiler and generated code execute within. Section two summarizes the structure of the server compiler. Sections three through section seventeen cover each phase of compilation in order. Solutions for specific language and runtime issues are described close to the compilation phase that addresses them. Last is a short description of phase costs and performance of generated code. Some names that occur in the compiler's source code are used within the text for readers who download the source under Sun's Community Source License [HS2.0]. The first occurrence of these names is emphasized to avoid confusion.

2. Runtime Environment

The server compiler generates code to execute within a runtime environment that also supports interpreter-only, Core, and interpreter plus client compiler execution. The runtime environment provides services which impact the performance of both compilation and the generated code. Several of the most significant features are: a single native stack per running thread for interpreting and executing compiled or native code, accurate garbage collection using card-marks, exception handling, efficient synchronization using a meta-lock [ADGKRW00] [BKMS98] [B98], class-hierarchy analysis, compilation events, on-stack replacement of interpreter frames with compiled-code frames, deoptimization from compiled code back to the interpreter, a compiler interface that supports compilations in parallel with garbage collection, and runtime support routines which may be generated at system startup.

The runtime generates the interpreter at startup using macro assembler templates for each bytecode and an interpreter dispatch loop [G99]. This provides assembly level instrumentation that collects counts at method entry and backward branches, type-profiles at call sites, and never-null object pointers for instanceof or checkcast bytecodes. Additional instrumentation has been implemented, e.g., branch frequencies, but is not turned on by default.

The runtime environment uses adaptive optimization to focus compilation efforts on performance critical methods [HU96]. These methods are identified using method-entry and backward-branch counters with additional heuristics that investigate the caller of a triggering method. When the combined method-entry and backward-branch counters for a method exceed the *CompileThreshold*, the runtime system determines which method to make the root of a compilation by

examining the call stack. If the caller frequently invokes the callee, the recompilation policy may decide to start compiling at the caller. This upwards traversal may continue multiple times relying upon the compiler to inline the path to the triggering method. Compiled code for the standard entry point is registered with the method object (*methodOop)* in a reserved field. At method invocation, the interpreter transfers control to compiled code when this field is not null. A different transition, on stack replacement, occurs when a method's combined counter exceeds the *OnStackReplaceThreshold* at a backward branch. The method is compiled with an entry point at the target of the backwards branch. The resulting code is registered with the methodOop, which contains a linked list of target bytecode index and compiled-code pairs. The runtime transfers execution from the interpreted frame to an on-stack-replacement frame and compiled code. The methodOop is used to cache other information as well, including the possibility that the compiler has refused to generate code for a method. A non-compilable method will always be run within the interpreter. This is used to support porting and debugging.

3. Compiler Overview

The server compiler proceeds through the following traditional phases: parser, machine-independent optimization, instruction selection, global code motion and scheduling, register allocation, peephole optimization, and code generation.

The compiler's intermediate representation (IR) is a static single assignment graph (SSA) based on the work done by Click and Paleczny [IR95]. It is used throughout optimization, conversion to machine instructions, scheduling, and register allocation. Operations are represented by nodes which have an ordered tuple of use-def edges pointing to nodes that produce values it requires. Both data and control flow are represented with explicit edges. When control-flow paths merge, e.g., after the body of an if statement, a *RegionNode* is built to merge the control edges. Values modified along these control paths are merged by a *PhiNode*, which has an edge to the RegionNode and a sequence of edges for the inputs. The order of these inputs corresponds to the order of control inputs to the RegionNode. In practice, the parser only builds regions and phis where control flow paths merge, not at the start of each basic block. In the resulting structure, values often flow directly from definition to use and not through control flow. This simplifies sparse analysis and optimization but additional structure is required to describe information resulting from control decisions. An example is the information that an object reference is null on one path after an explicit null-check and not-null on the other. To add information to a type we insert a new definition *CheckCastPPNode* during parsing. This retains the information until the value is merged into a phi. In addition to null, not-null, and the object hierarchy, the type system handles primitive types and the control type. During a non-optimistic analysis phase, a "TOP" control input immediately signals that this control flow path is dead.

4. Parser

The parser's first pass over the bytecodes identifies basic blocks and their predecessors. The second pass visits basic blocks and for each block translates each bytecode to the compiler's IR. The basic block parsing order will visit a block whose predecessors have been parsed before a block with unparsed predecessors whenever possible. This makes type propagation during parsing as beneficial as possible without requiring backtracking. After parsing has completed, def-use edges are constructed in a batch pass to eliminate any useless code produced by the parser. These def-use edges are maintained during optimization until a machine-specific representation is constructed.

Since there are bytecodes at both low and high levels of abstraction, the number of nodes in the IR used to represent a bytecode varies widely. The bytecode *iload_<n>* does not require any IR construction because the parser tracks the status of locals, expression stack, and monitors in a *JVMState* object. We translate this by updating an entry in the current JVMState object such that its top of the expression stack contains a pointer to the indicated local. Other low-level nodes require the construction of one or more IR nodes. An example of this is *iadd* where two entries are popped off the JVMState expression stack and *AddINode* is built. This node has a null control edge and two inputs, pointers to the nodes that were popped off the expression stack. Before pushing the result onto the JVMState's expression stack the AddINode is optimized. The optimizations applied during parsing, *Ideal*, *Value*, *Identity*, *GVN*, are a subset of those applied during post-parse optimization since def-use information is not available. Ideal canonicalizes the node structure locally and upwards along use-def edges. It may reorder the inputs to a commutative operation like AddINode to support value-numbering or even construct replacement nodes as in the conversion of (constant + (constant + variable)) to (variable + (constant + constant)). Value performs abstract interpretation to produce a type for the result of this node. This transformation constant folds (constant + constant) to (constant). Identity recognizes when the result is equivalent to one of its inputs and returns that input instead. This transformation cleans up (variable + zero). Finally, global value numbering checks a

hash table to see if this node's value is already recorded. If it is, the node that previously produced it is returned and the newly constructed node is discarded. For new values we record the pair *<AddINode::hash_code(), &AddINode>*.

When parsing a higher-level bytecode, instanceof, we apply the transformations to each node in turn. The generated IR first checks if the object is null. When the result of this check can be determined at parse-time, by inspecting the compiler's type for the input, the never taken branch will not be constructed. For the not-null branch, we generate a fast/slow path idiom. A fast/slow idiom is a diamond-shaped control flow graph (CFG) where the expected case is to do quick checks along the fast path to confirm a result. When some portion of the fast path fails, control transfers to the slow path which covers all remaining cases. For *instanceof*, the fast path checks a two-element class cache to see if this object's class has recently been verified as being the desired instance. If this fails, the slow path calls a runtime routine that checks the subtype relationship and updates the cache as necessary. We did not implement a failure cache as preliminary investigation did not confirm a performance improvement for common programs.

Object allocation provides another example of a fast/slow idiom. The fast path has an atomic test and set of the heap top. If the new heap top is safe, fast-path allocation is complete. If it has exceeded the upper limit for this memory space, execution continues to the slow path and the thread is stopped at a safepoint for garbage collection. Allocation of objects with finalizers is also done by the slow path since finalizer registration is done within the runtime system. Additional fast/slow path idioms are generated for checkcast, object array store, other types of object allocation, and division by zero.

5. Uncommon Traps

In HotSpot, we compile methods that have crossed a threshold. In most cases any necessary class initialization or class loading has already been done by the interpreter which handles all initialization semantics. We investigated having the generated code handle class initialization properly and discovered that it is too rare. Instead the compiler generates an uncommon trap, a trampoline back to interpreted mode, when it compiles a reference to an uninitialized class. The compiled code is then deoptimized and it is flagged as being unusable. Threads entering the method are interpreted until its recompilation is finished. As a side effect, field offsets are always known so short-form addressing modes can be used without backpatching.

6. Call Sites

All *invokeX* bytecodes are parsed in a single function, *do_call,* to keep related pieces of code together. do_call starts by checking that the destination method is loaded, its holder is initialized, and the return value is loaded. Failure at this point results in generating an uncommon trap instead of the call. After checking these safety conditions, do_call inlines the callee or generates one of two distinct calling mechanisms, static or dynamic. *Static* calls dispatch directly to the verified entry point of a method and are used for static calls and non-inlined virtual calls that have only one receiver. *Dynamic* calls dispatch to the unverified entry point of a method and are preceded by an instruction that places an inline cache holder in a register. The unverified entry point compares the dynamic receiver's class to the class in the inline-cache. If the unverified entry point fails then control transfers to a runtime routine to patch the call-site. Since any non-final virtual method may be invoked using this technique, the inline-cache register is not used for passing normal arguments. Inline caches have been used for dynamic object-oriented languages since Smalltalk 80 [DS84].

Virtual and interface calls are examined to determine if there is only one receiver. First with class hierarchy analysis (CHA) and second with receiver profiling done during interpretation. If CHA identifies a single receiver the callee will either be inlined or called as an optimized virtual call using the static call mechanism. In these cases we record that this method is dependent upon the class hierarchy below the receiver class. A later class load could provide another implementation of the target method requiring deoptimization of the compiled code [HCU92]. The mechanism is described in the next section. Inlining recursively invokes the parser on the target method to generate an intermediate representation of the callee with the type information at the call-site available as context. If CHA does not identify a single target but the receiver profiling record only has a single class as receiver, we generate runtime verification code and inline the target method. Unlike the approach taken by IBM [SOTYKIKN00], the failure path does not form a diamond containing the inlined method on one side and a virtual call on the other since the merge point would destroy the precise information gained by the explicit check. Instead the failure path generates an uncommon trap, resulting in deoptimization and recompilation of the method if the receiver changes. This is a modification of uncommon-branch-elimination [HU96] which normally clones the rest of the method for the infrequent path. A CheckCastPPNode is inserted to force the type of the receiver to the profiled type, eliminating the need for such checks later in the method.

Although inlining provides the most opportunity for optimization, making a virtual call has its own rewards. A non-inlined virtual call does an implicit check that the receiver is not null. After these call-sites a CheckCastPPNode forces the callee-type to not-null.

Exception paths are connected after either inlining or creating a call node. If inlined, all exceptions thrown by the callee are checked against a list of handlers in the caller. When a matching handler is available, the generated code verifies correctness and jumps directly to the handler. If the correct handler can not be identifed, e.g., caller does not have a handler, the callee's exception path becomes a rethrow and the runtime system finds the handler. Since exceptions are considered infrequent, exception regions are not registered during normal execution as in [SOTYKIKN00]. Although this reduces the impact on non-exception throwing code, the runtime may have to traverse the stack to find a handler.

7. Deoptimization

If class loading invalidates inlining or other optimization decisions, the dependent methods are deoptimized [HCU92]. Threads currently executing in the method are rolled forward to a safepoint, at which point the native frame is converted into an interpreter frame. The invalidating class load is not visible to the executing thread until it has been brought to a safepoint. Execution of the method continues in the interpreter.

Deoptimization, therefore, requires that we can regenerate the interpreter's JVM state at various points in the program. This is very much the same information required to debug optimized code. We do this by recording the exact JVM state as inputs to safepoints and procedure calls. The entire JVM state is thus "live" into the safepoint. We then allow the optimizer and register allocator to do their best with this JVM state information. Finally, during code emission we build a table mapping the JVM state to the final resting place of the still-alive JVM state information. The result is well optimized (but not perfectly well optimized) code with exact JVM semantics at specified program points. The result of keeping JVM state around is that some extra values remain "live" longer. These long-lived values generally spill to the native stack frame and have no more cost than an extra store in the final code.

8. Optimizer

Since parsing attempts to visit a block after all of its predecessors, it is at loop headers that further progress from the set of parse-time optimizations is possible. To avoid revisiting nodes that are already at a fixed point, the parser produces a worklist containing nodes that may benefit from additional transformations. The results of prior abstract interpretation are also provided as input to the next phase, iterative GVN. *PhaseIterGVN* applies a pessimistic sparse iterative algorithm until it reaches a fixed point. This is done after building Def-Use edges and then forward propagating changes from Ideal, Value, Identity, and GVN transformations for all nodes on the worklist. As described, this will only reach a fixed point for optimizations that inspect their immediate definitions since an intermediate node might not change value stopping the iteration. To ensure a fixed-point for transformations that inspect a node's grandparents, the iteration inspects grandchildren for additional nodes that must be added to the worklist. Optimizations that use def-use information to inspect their children are not required for program correctness and are not guaranteed to reach a fixed-point.

The next phase, *PhaseIdealLoop,* iterates until no "major" control-flow changes occur. Major changes occur in the following cases: a loop becomes dead; an 'if' is removed because it is duplicated by a dominating test; an 'if' is cloned upward through a region. The cloning only occurs when at least one cloned "if" will fold away. This optimization is enabled by default starting with the Solaris Kestrel Release. During this phase loop peeling removes null checks from inside the loop.

The last pair of machine-independent optimizations re-apply constant propagation and iterative GVN until a fixed-point to perform global dead code elimination. The constant propagation algorithm is optimistic sparse conditional constant [WZ85] where forward propagation starts with each value initialized to TOP. Constant propagation also places some grandchildren on the worklist to support Value calls that examine their grandparents. This occurs for a *CatchNode* following a call, which defines exception paths, since it examines not only the value of the call node but also the value of the receiver parameter. The standard return path following the call is not enabled until after it is proven that the receiver input may not be null. Before a call node is pushed onto the worklist its uses are examined to find the CatchNode and push it onto the worklist. A similar investigation is performed before placing a RegionNode onto the worklist since its value might not change when a control-flow input changes, but PhiNodes that depend upon it must update their value if a control-flow path becomes dead.

After constant propagation has reached a fixed point, nodes that have been identified as constant are replaced in a recursive use-def traversal that updates

both use–def and def–use information. During this traversal any control dependence information maintained by cast–nodes, e.g., CheckCastPPNode, is transferred to memory operations that use the address. This allows safe removal of unnecessary cast–nodes in the next GVN pass. In addition, a worklist is generated so the following iterative GVN pass does not examine the entire program. The worklist is initialized with nodes that use the newly discovered constants. It also contains "if", "region", and "loop" nodes plus conversions from integer to boolean and pointer to boolean. These nodes are likely to benefit from the optimistic type information generated during constant propagation. In particular, null checks benefit from improved null/not–null type information.

9. Instruction Selection

Translation from machine–independent instructions to machine instructions uses a bottom–up rewrite system, BURS **[PLG88]**. This is done before placing instructions into basic blocks so that the selection is unencumbered by block boundaries. Before translation the sea–of–nodes that exists during optimization is divided into possibly overlapping subtrees by labeling each subtree's root node. Subtree root candidates are nodes with multiple users and nodes that may not be duplicated because of side–effects, e.g., the atomic add idiom for memory allocation. By convention a root produces its result in a register so the value may be shared without recomputation. Shared nodes that are not labeled as roots are reached by multiple paths producing duplicate computation. Root selection is preferenced to only duplicate address expressions and other idioms that the machine architecture can subsume into one instruction. Nodes that are not translated to machine instructions, e.g., PhiNodes, are recorded in a *dontcare* array.

Machine specific nodes for each subtree are generated in two steps. First a postorder walk along use–def edges starts from a selected root and visits all children recursively unless the child is a root, is not matchable, or has a different control input than the walk. When visited, the node and the possible translations of its children, recorded in a *State* object, are passed to a deterministic finite automata (DFA). The DFA records the lowest cost instruction for each possible result in a new state vector for use when the node's parent is visited. This process provides optimal instruction selection within the constraints of subtree selection, accurate costs for machine instructions and operands, and available instructions in the architecture description. The result of the postorder walk is a binary state tree identifying the least–cost instruction to generate at each level. Recursively traversing a state tree, the preorder visit generates the lowest execution–cost machine instruction for that level. The postorder visit attaches the instructions generated for its inputs. Additional edges are inserted for machine–independent information including memory and control.

10.Global Code Motion

The framework used to place instructions into basic blocks is a modified version of that described in **[C95]**. Our first step is to build a control flow graph skeleton of basic blocks. Virtual calls for *is_block_start* and *is_block_proj* identify nodes that start and end blocks respectively. Since the sea–of–nodes representation does not contain a block start or region node for each block, one is constructed where necessary. Next we compute dominator information for each block using the algorithm by Lengaur and Tarjan **[LT79]**. Block frequency estimates are generated using two forward passes. The first does forward propagation from method–start in reverse postorder scaling loop header frequency upwards by ten and splitting frequency at ifs using the probabilities recorded during parsing. The second pass updates block frequencies using information from all predecessors instead of only the depth first path from start. This propagates information around loops. An additional update done during the second pass sets the block frequency of any block ending with a halt instruction to medium–rare.

We assign instructions to blocks in three stages. The first stage identifies the earliest legal block for each instruction. This is done recursively by examining the earliest legal block for each input and finding the one which is deepest in the dominator tree. The base cases are provided by the nodes pinned into the basic block skeleton which include: Start, Region, Phi, Goto, and Return. The second stage identifies the latest legal block for each instruction during a depth–first walk over def–use edges. The def–use information used during optimization was lost when new machine–specific nodes were generated so it is rebuilt at the start of scheduling. The depth–first walk's postorder visit to a node computes its latest legal placement as the least common ancestor of all uses. If the node is a store to memory we insert any required anti–dependence edges, an edge from the store to a load that must precede it. Control–flow and memory alias analysis are used to identify loads and stores that are independent, known to access distinct memory locations, allowing more scheduling freedom. This late insertion of anti–dependence edges requires that the depth–first walk visit all children which may be loads after their siblings. A store can fail to schedule if a load that must precede it has been placed into too late a block. Stage three selects a block between the earliest and the latest legal location by walking up the dominator tree and

identifying the block with least frequency. This is done during the postorder visit after computing the latest–legal block and inserting required anti–dependence edges.

11.Local Scheduler

The local scheduler orders instructions internal to a basic–block by selecting from a worklist of ready nodes and placing the selected instruction at the next available position. An instruction is placed on the ready list when all of its inputs within the block have been selected. Inputs from instructions in different blocks are not considered. The select routine computes a score for each ready instruction and returns the one with the highest score. The initial score is determined by several heuristics (e.g., nodes that only have uses outside the current block start with a lower score to encourage scheduling them late.) An instruction that stores to memory or the stack is given a higher initial score to free any registers its inputs may be using. The initial score is biased to delay loads and prefer instructions with many inputs.

12.Global Register Allocator

The register allocator is structured in three phases. The first phase converts the SSA based CFG into non–SSA form (SSA structure is also retained at this point for use in the allocator.) The second phase is the main allocation loop, which is fundamentally a Briggs–Chaitin Graph Coloring Global Register Allocator with a number of refinements for both allocation speed and code quality. The third phase is a cleanup phase which manifests and cleans up the instructions generated by the allocator, and records Garbage Collection information.

Our allocator has a number of special features. We use our allocator to perform the normal duties of a calling convention module. That is, our calling convention merely specifies the legal registers for values which flow into calls, and we allow the allocator to generate any necessary instructions to move the values to the appropriate locations prior to the call. In addition, we use the allocator to allocate locations on the hardware stack frame as if they were registers, thus minimizing frame size. We also replace standard stack based spilling for live ranges which fail to color, with a special form of live range splitting. Finally, we use the allocator to generate the information necessary to perform garbage collection during execution of the compiled code.

13.Reverse SSA Transform

Since our IR uses an SSA based structure, we must have a phase in the compiler which converts from the SSA form of the program to the non–SSA form. We have chosen to integrate this phase with the Register Allocator. We perform a pass over the CFG which inserts "virtual copies" at phi node sites. These "virtual copies" are virtual in the sense that we do not fully insert the nodes into the graph until later. We then perform global live analysis on the annotated CFG, and gather information about the legal set of registers for each live range. Using that information, we construct an "interference graph" (IFG) for the method. The IFG represents the analysis of which live ranges "interfere", that is, could compete for the same registers.

We then use the IFG to perform a pass of copy coalescing. This pass is aggressive in the sense that it could remove a copy which we will later have to reinsert, but it is pessimistic in the sense that it assumes that copies are necessary, and attempts to prove that they are not. This special coalescing pass only works on the virtual copies inserted during the reverse SSA transform. Any virtual copies which remain after the aggressive coalescing pass are then converted to actual copy instructions in the CFG. The CFG then contains the non–SSA version of the method, while the normal IR version of the method is also retained with its SSA form and phi nodes. This dual representation allows us to perform SSA based analysis and optimizations while ensuring that we allocate all the copies necessary to restore normal naming. Finally, we execute a pass over the CFG which inserts extra uses of object pointers which serve as the base of a derived pointer. These extra uses are added at safepoints in order to ensure that the base object pointer values are still live and available at the safepoint location so that the garbage collector can inspect and relocate them.

14.Main Allocation Loop

The principal work of register allocation is performed in an iterative loop. We perform live analysis, gather the legal register sets for the live ranges, and build an IFG for the method. Then we perform a conservative pass of copy coalescing (i.e., we only coalesce where we can prove that removing the copy will not force another interfering live range to fail to color.) This pass is followed by the standard simplification and color selection phases of a graph coloring allocator [BCT94]. Finally, if we fail to color all live ranges, we perform a transform which splits any live range which failed to get a color into a series of smaller live ranges connected by copies. Then the process repeats until a complete coloring is found.

Our implementation of a graph coloring allocator has a few special features designed to improve the speed of the allocator as well as to improve the quality of the

code generated. During the construction of the IFG, we transform the legal register set for live ranges which interfere with what we call "bound" live ranges. Bound live ranges are those which are restricted to a single predefined register. A typical example of this is found in the arguments to a call which are passed in a register specified by the calling convention. Since these live ranges must color into that particular register, we remove that register from the legal register set of all live ranges which interfere with the bound live range, and which are not also bound. This significantly reduces the number of interferences in the IFG, and speeds the computation of the IFG.

In addition, we also track a metric of "register pressure" during IFG construction. Each basic block is tagged with a value which indicates that the block has either high or low register pressure, and where in the block a transition from low to high pressure takes place. We use this information during our live range splitting pass. This pass performs a reaching definitions data flow propagation, inserting copies into any live range which failed to color at any point where that live range crosses a high register pressure boundary. The copies are tagged with a special legal register set which allows them to color onto the stack rather than in the hardware registers. This splitting has two benefits. It inserts fewer copies than splitting at every use and definition which speeds up copy coalescing and live analysis. In addition, since we split with a copy, the new live ranges could still color into registers even though the original unified range could not. This coloring would include some register to register copies, which are far less expensive than loads and stores. These special features are more completely described in [CPV01].

15.Cleanup

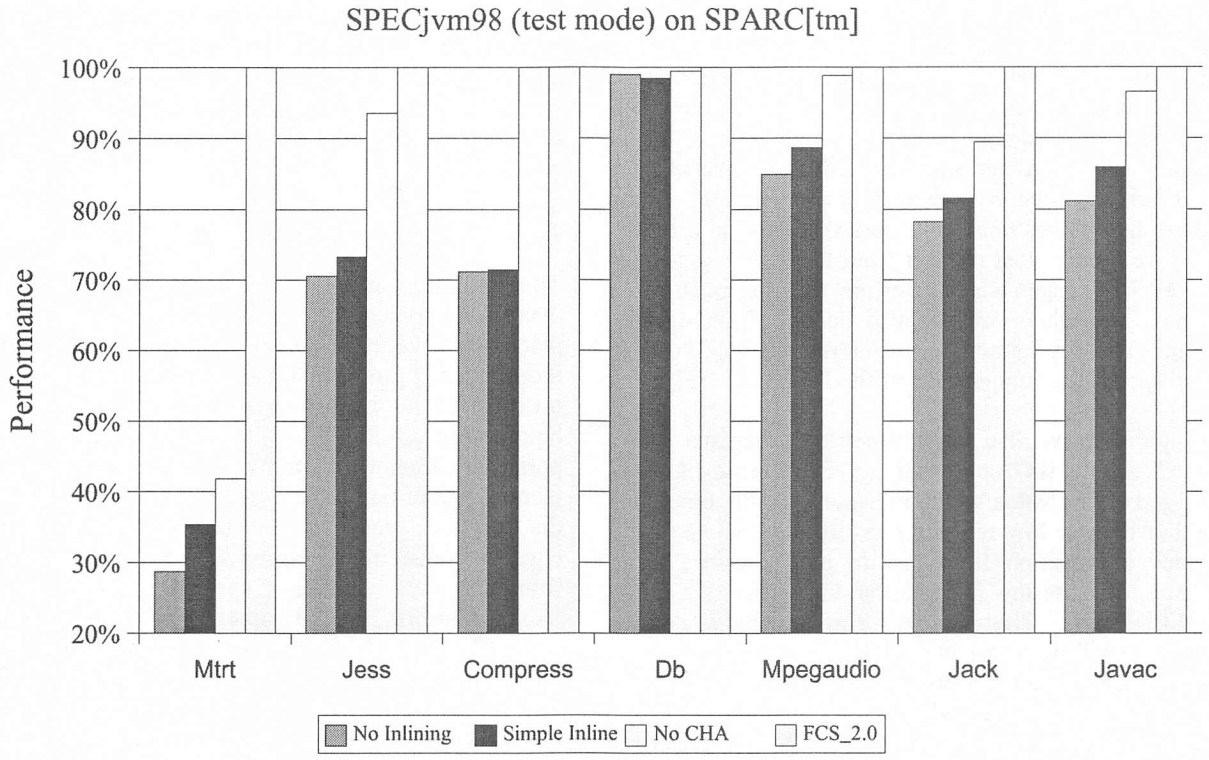
The third phase of the allocator performs several cleanup and bookkeeping functions. We remove unnecessary copies using a variant of register tracking to determine that a value that was split can actually live in a single location given the actual coloring assigned to the method. Then we gather and output the information necessary for garbage collection. This includes the location of all object pointers which are alive across a safepoint, as well as the location of all values which are callee saves in the method, and all necessary computation information for derived pointers which have an object pointer as their base. Finally, we perform a pass which transforms our special copies which were inserted during live range splitting and which failed to be colored into a hardware register (i.e., they were colored onto the hardware stack frame), with the appropriate load or store instruction. This pass also performs a special optimization on CISC architectures which attempts to combine these loads or stores into a direct memory reference with either a user or definer of the value being copied, which we refer to as "CISC spilling". This process produces the final version of the CFG which we use for output in the final phase of compilation.

16.Peephole Optimizer

This optimization inspects each sequence of adjacent instructions to determine if the instructions may be replaced by a better sequence [ASU86]. The machine independent portion of this optimization visits every instruction, and invokes its machine-dependent peephole optimization. This is a virtual call built by our portability framework from information in the architecture description. The operands of instructions in the original sequence can be required to satisfy equality constraints to ensure the safety of the transformation. A commonly applied peephole on the IA32™ platform is replacement of "MOV dest_reg, src_reg" followed by "INC dest_reg" with the single instruction "LEAL dest_reg, dest_reg + 1". The equality constraint requires that the destination of the move and the register being incremented are identical. The current implementation restricts the replacement to a single instruction.

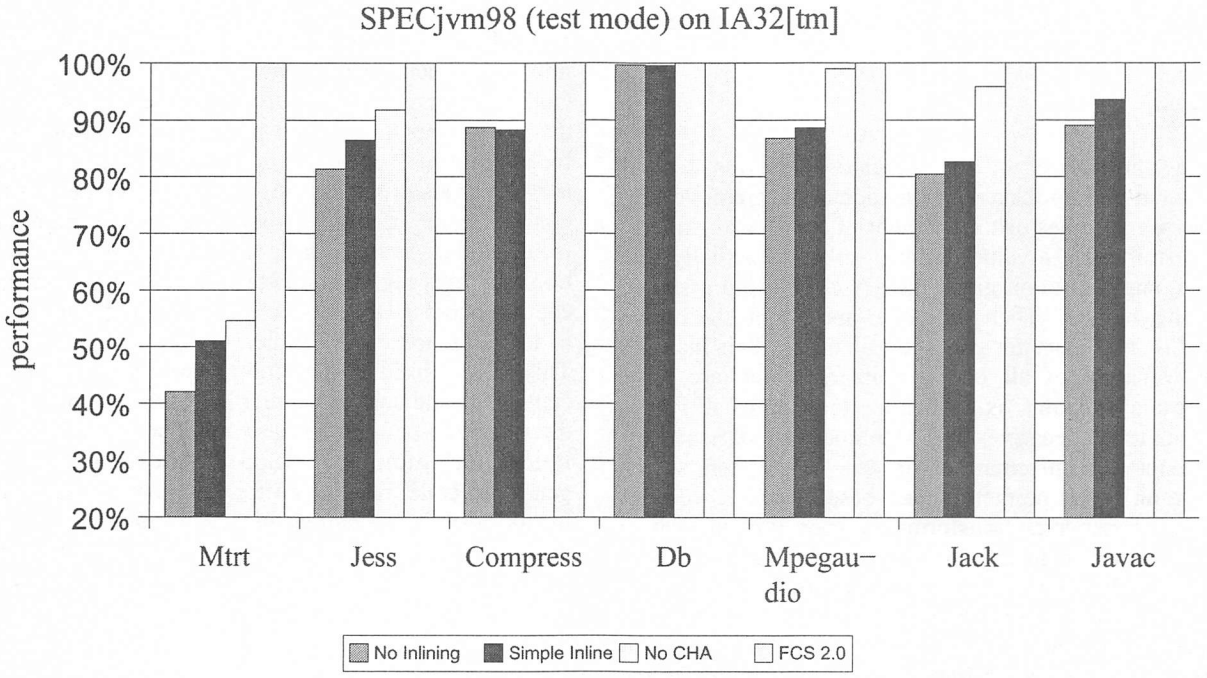
17.Code Generation

In addition to executable machine code, the code generator also provides oopmaps, debug info, exception tables, relocation information, and an implicit-null check table for use by the runtime system. All of this information is associated with one or more native-code offsets from method entry. Oopmaps and debug info are associated with the offset to their safepoint. Oopmaps are generated during register allocation and the code generator simply packages this information for the runtime. Safepoints at which a deoptimization may occur also record debug info describing either the constant value or native storage location for monitors, locals, and expression stack entries. The storage location may be a register or a stack frame offset. An exception handler table at each call-site maps the bytecode index for each handler to its handler's offset. This table is used by the runtime system to vector exceptions to the correct handler in generated code when the transition can not be determined at compile-time. Relocation information supports movement of the generated code from the buffer in which it is generated to its installed location in the *CodeCache*. The method's implicit null check table contains each offset at which an implicit null check may occur, paired with the corrsponding handler's offset.



As a preliminary step to generating code and runtime-system tables, we calculate the size of the executable by sizing each instruction. While this is straightforward on RISC architectures, time pressure and engineering simplicity led us to generate size information for CISC architectures by doing code generation twice. The first pass generates code for every instruction in a temporary buffer to calculate instruction size, basic block start/end, relocation size, reserve space for constants, and the return address for calls.

With all sizes and offsets computed, we give branch



instructions their offsets. If the target architecture provides a shorter branch instruction format for local displacements, a replacement can be generated using the virtual method *short_branch_version* provided by the portability framework. Final code emission is done using the same *MachNode::emit* virtual used for the earlier size calculations, which emits raw machine object code into a buffer which is returned to the run-time system as the primary result of the compilation. The compiler interface relocates the generated code to the CodeCache and registers it with the methodOop.

18.Floating Point Precision

On the IA32™ architecture HotSpot normally executes with the FPU control word set for 53-bit precision and round to nearest. This means that when executing a non-strict method the result of each float operation must be restricted to 24-bit precision, generally by spilling to memory or the stack. Before performing instruction selection, we count the number of float, double, and invoke actions. When the following heuristic is satisfied we switch to 24-bit floating point mode: (#doubles == 0 && #floats > 32 && #floats > 10 * #calls). This mode modifies instruction selection such that individual floating point operations do not have to be spilled to the stack until assigned. In addition, at method entry and upon return from any outgoing call the FPU control word will be set to 24-bit precision. Before every method exit point and every outgoing call the FPU control word is returned to the system's default 53-bit precision.

19.Experimental Results

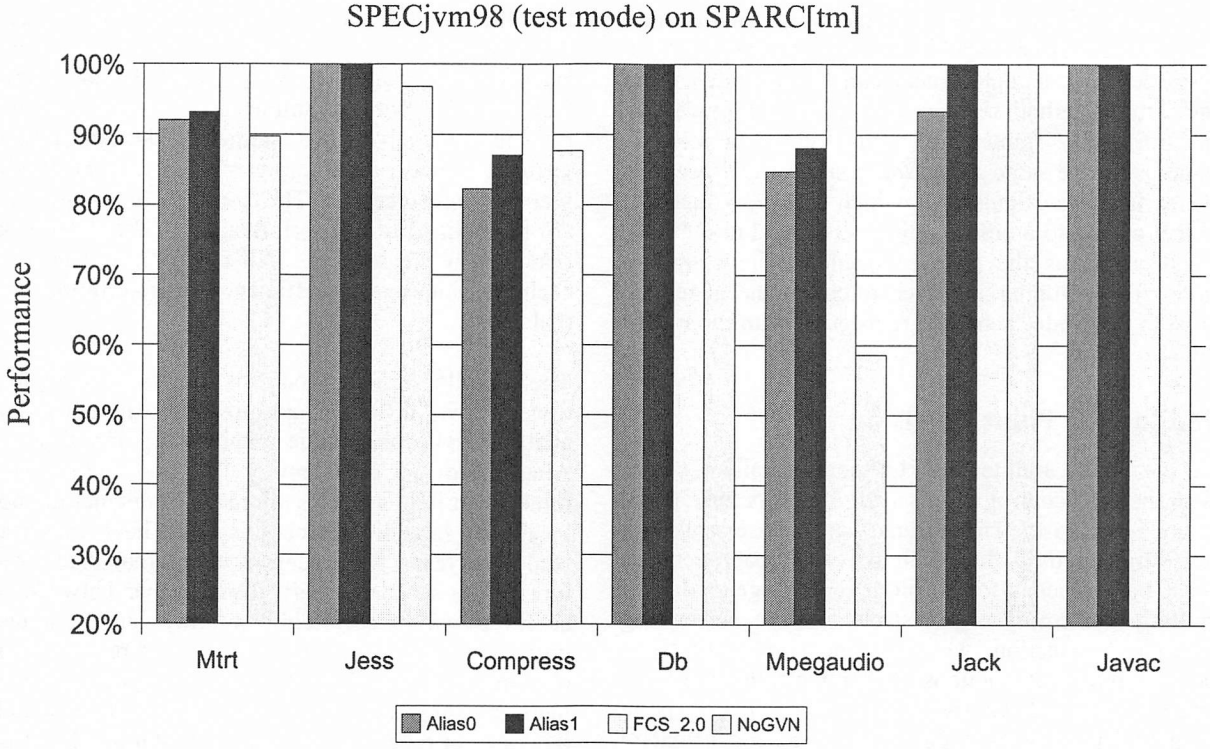
The server compiler applies both object-oriented optimizations, such as class hierarchy based inlining, and traditional compiler optimizations, such as memory alias analysis and global value numbering. The results for two target platforms are presented in bar charts with one column for each optimization per benchmark. Each column is normalized to the performance of the downloaded sources. The first pair of charts illustrates four variations in inlining; the second pair illustrates two compiler optimizations. These benchmarks are from the SPECjvm98 client suite [SPEC98]. The results are from test mode execution, the benchmarks are run as a group from the command line, not in SPEC-compliant mode. The SPARC platform is a Sun Ultra 60 with the process bound to one of the 450MHz. UltraSPARC-II processors. The IA32 platform is a Dell Dimension XPS B866r with 256 MB memory.

The inlining tests cover four variations. The first, no inlining, turns off all inlining during compilation. It also turns off the possibility of starting a compile at the caller when a method's counter reaches the activating threshold. The second only allows inlining of accessor methods. The third uses the default inlining heruristics but does not inline virtuals since there may be multiple receivers. In addition, it does not inline a virtual call when the interpreter's call-site profile has seen only one receiver. The results of these three tests are normalized to the performance of the default parameters in the sources. All of the tests use inline caches to improve the efficiency of non-inlined virtual calls.

The compiler optimization tests illustrate the application of two different optimizations, memory alias analysis and global value numbering. The first test, Alias0, does not use memory alias analysis to determine when field accesses are made to distinct memory locations. In addition, accesses to the heap are serialized with respect to accesses to internal data structures. The second, Alias1, distinguishes between accesses to an object field, an array length, a class pointer in the object header, and a VM private memory space. The default analysis further partitions memory using the class holder and offset for a field, the element type for an array, and additional distinctions for internal data. The fourth column for each benchmark uses the default alias analysis but disables global value numbering except for control structures and comparisons (which is required by implementation constraints).

Inlining accessors shows a small improvement on all benchmarks except compress and db where the differences are not significant. Larger improvements occur from more extensive inlining even without inlining virtual calls. On SPARC this provides a 20% improvement for JESS while on IA32 it provides a 12% improvement for JACK. Inlining based on class hierarchy analysis almost doubles the performance of MTRT, but provides only minor imprevements to four benchmarks on SPARC and five on IA32. On most of the benchmarks, enabling inlining helps SPARC more than IA32. The heuristics that control this optimization are tuned for each platform, and the larger register set on SPARC allows additional inlining without hurting register allocation.

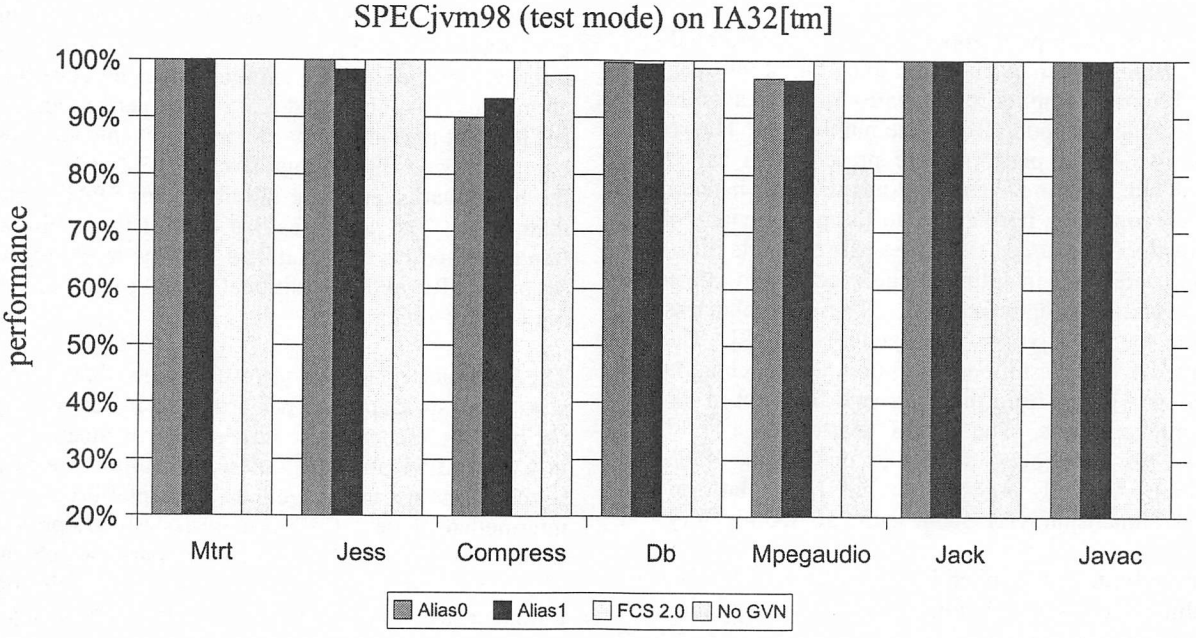
The compiler optimization results show that memory alias analysis improves performance by up to 17%. On both architectures the largest improvements occur in COMPRESS and MPEGAUDIO while three of the benchmarks are unchanged on both platforms. The intermediate level of alias analysis, Alias1, provides small improvements for those benchmarks which vary. The last column for each benchmark shows the performance when not using global value numbering, an optimization which commons together equivalent



subexpressions. One common source for these expressions is array accesses. The largest benefit on both platforms is from MPEGAUDIO which does have frequent array accesses. Global value numbering provides significant performance improvements on SPARC, it has less impact on IA32. While the duplicate expressions are present after optimization on both platforms, but simple array access expressions can be folded into a CISC load or store when converting to machine instructions.

20.Experiences

The server compiler emphasizes the quality of generated code, relying upon the 80/20 rule to spend more time optimizing frequently executed code. This costs compilation time with the approximate breakdown being 14% parser, 20% optimizer, 6% instruction selec-



tor, 7% scheduler, 49% allocator, 4% code generator. However, the resulting performance helps achieve a SPECjvm98 score of 81.4 on a Dell Dimension XPS B866r with 256MB memory [Spec98]. Source code for the Java HotSpot™ Virtual Machine is available for download under terms of Sun's Community Source License [HS2.0].

21.References

[ADGKRW99] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. Ramakrishna, and D. White. An Efficient Meta-Lock for Implementing Ubiquitous Synchronization. In *Proceedings of the ACM SIGPLAN '99 Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '99)*, pages 207-222, November 1999.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Co., Menlo Park, CA (1986).

[BCT94] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428-455, May 1994.

[C82] G. J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices 17*, 6 (June 1982), 98-105. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[IR95] C. Click, M. Paleczny. A Simple Graph-Based Intermediate Representation. In *ACM SIGPLAN Workshop on Intermediate Representations (IR '95)*, pages 35-49, Jan. 1995.

[C95] C. Click, *Combining Analyses, Combining Optimizations.* Ph.D. thesis, appendix, Rice University, 1995.

[CPV01] C. Click, M. Paleczny, C. Vick. Interference Graph Trimming. Submitted to *ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, 2001.

[BKMS98] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation,* pages 258-268, June 1998.

[B98] L. Bak, presentation on the HotSpot JVM, Panel: The New Crop of Java Virtual Machines. In *Proceedings of the ACM SIGPLAN '98 Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '98),* pages 179-182, October 1998.

[G99] R. Griesemer. Generation of Virtual Machine Code at Startup. In *OOPSLA'99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design.*

[HCU92] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 1-12, Jun, 1992.

[H93] U. Hölzle. A Fast Write Barrier for Generational Garbage Collectors. In *OOPSLA'93 Garbage Collection Workshop*, pages 1-6, Oct, 1993.

[HU96] U. Hölzle and D. Ungar. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages. In *ACM Transactions on Programming Languages and Systems*, 18(4):355-400, Jul, 1996.

[HS2.0] Java HotSpot™ Virtual Machine 2.0. June 2000. http://www.sun.com/software/community-source/hotspot/download.html.

[IKY98] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu and T. Nakatani. Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler. In *ACM 1999 Java Grande Conference.*

[LT79] T. Lengauer and R. E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. In *ACM Transactions on Programming Languages and Systems*, 1(1):121-141, 1979.

[PLG88] E. Pelegrí-Llopart and S. L. Graham. Optimal code generation for expression trees: an application BURS theory. In *Proceedings of the Conference on Principles of Programming Languages (POPL'88)*, pages 294 - 308, 1988.

[Spec98] SPECjvm98 Benchmarks. June 1, 2000. http://www.spec.org/osg/jvm98.

[SOTYKIKN00] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Java Just-in-Time Compiler" *IBM Systems Journal*, 39(1):175-193, Jan, 2000.

[THS98] O. Traub, G. Holloway, and M. Smith. Quality and Speed in a Linear–scan Register Allocator. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation,* pages 142–151, June 1998.

Can a Shape Analysis Work at Run-time?

Jeff Bogda and Ambuj Singh
Department of Computer Science
University of California
Santa Barbara
{bogda,ambuj}@cs.ucsb.edu

Abstract

A shape analysis is a whole-program analysis that can identify run-time objects that do not need to be placed in the global heap and do not require any locking. Previous research has shown that these two optimizations can speed up some applications significantly. Unfortunately, since a shape analysis—like any whole-program analysis—requires *a priori* knowledge of the complete call graph, it has not been implemented in a JVM, which essentially builds the call graph as a program executes. In this paper, we adapt an efficient shape analysis to be incremental so that it can analyze an executing program. We investigate trade-offs regarding three approaches to performing the analysis inside a JVM and report results on a number of applications. Our measurements suggest that such an analysis may be viable if it uses results of previous executions and if it delays the initial analysis until the end of the first execution.

1. Introduction

A shape analysis [17] is a static whole-program analysis that conservatively predicts the connectivity of heap objects. It proves useful for escape analyses because it identifies objects that may "escape" a method and that may be accessible to more than one thread. Such knowledge steers optimizations such as synchronization elimination and stack allocation.

Unfortunately, shape analysis, as described heretofore, ignores the true flavor of Java programs. The Java programming language prides itself on its dynamic loading and binding, yet a shape analysis requires *a priori* knowledge of all classes. This requirement stems from the fact that it must know all potential targets of a call site. Without the knowledge of potential targets, it is forced to be overly conservative. The many programs that use dynamic loading or that rely on dynamic program attributes—such as the classpath variable—will not benefit from optimizations dependent on the results of a whole-program shape analysis.

Even without Java's dynamic features, an optimizer faces a difficulty. It cannot express, in bytecode, the removal of lock operations or the stack allocation of objects—exactly those optimizations that the shape analysis enables. Researchers sidestep this problem either by annotating the bytecode with suggestions to the JVM or by translating the program into a language in which these optimizations are expressible.

The only true solution to these limitations is to perform the shape analysis at run-time. By operating while the analyzed program executes, the analysis can *observe* (as opposed to calculate) the classes the JVM dynamically loads. More importantly, it can observe the targets of a call site, yielding more precise results than an off-line analysis would. Last, it is not restricted to optimizations expressible in bytecode and can perform optimizations in a JVM-dependent manner.

Despite these advantages, a dynamic shape analysis faces several difficulties. It incurs a run-time cost and must work with incomplete information. Since it is by nature a whole-program analysis, it must build upon and modify previous results as new information arrives. Consequently, a previously optimized object may no longer be optimizable, requiring the optimizer to undo optimizations before an erroneous execution results.

To address these difficulties and trade-offs, we present and evaluate three ways to perform a shape analysis at run-time. The first approach begins the interprocedural analysis as soon as a program starts to run. The second delays the interprocedural analysis until the run-time system has seen a portion of a program's execution. Finally, the third approach reuses analysis results from previous executions. We discuss the advantages and disadvantages of each strategy.

In short, this paper offers:

- An incremental version of an efficient shape analysis;
- Experimental results illustrating the inherent difficulties of employing a shape analysis dynamically;
- A comparison of three approaches to performing the incremental analysis; and
- Insight into making the analysis viable.

```
for each strongly connected component (SCC) in rev. top. order
  for each method m in the SCC
    analyze m intraprocedurally
    for each call site s in m
      for each target t of s
        if t and m are in the same SCC
          unify actuals of s and formals of t
        else
          propagate from t to s
```

Figure 1. Static shape analysis algorithm.

After outlining an efficient whole-program shape analysis (Section 2), we adapt it to be incremental (Section 3). In doing so, we recognize that the analysis can classify objects with varying degrees of locality. In some cases, it can guarantee that an object will be local to a thread regardless of future program paths and class loadings. In Section 4 we show the results of an empirical study that compares the aforementioned ways to perform the analysis. Last, Section 5 presents related work, and Section 6 presents conclusions, two related open problems, and future work.

2. Whole-Program Shape Analysis

This section sketches a conservative, but efficient, version of a whole-program shape analysis based on the analyses described in [4] and [10]. Section 3 adapts the algorithm presented here to be incremental. Since the emphasis of this paper is on performing a shape analysis at run-time, not on the efficacy of the analysis itself, we omit strategies one can use to improve precision. For a complete description of the problem, we refer the reader to [1,3,4,5,6,10,16].

A shape analysis is an interprocedural data-flow analysis that approximates the run-time structure of heap objects and identifies objects potentially reachable from a static field. Figure 1 presents the algorithm at a high level. The results of this analysis dictate when it is safe to perform certain optimizations.

We view the results of a shape analysis as a graph. A node in the graph is an abstraction of one or more run-time objects. An edge in the graph represents an instance field dereference and is labeled with the name of the field. The analysis associates each program variable with a node in the graph and connects nodes to reflect the structure of the heap. In the end, if the analysis has associated two variables within a method with distinct nodes, it guarantees that these variables can never reference the same object at run-time. Furthermore, if the analysis has associated a variable with a node that is marked *shared*, the analysis believes

```
import java.util.*;

public class Example
{
 public static void main( String[] args )
  throws ClassNotFoundException, InstantiationException,
    IllegalAccessException
 {
  String className = args[0];
  Class theClass = Class.forName( className );
  List listImpl = (List)theClass.newInstance();
  String type = args[1];
  test( listImpl, type );
 }

 private static void test( List list, String element )
 {
  if( element.equals( "int" ))
   for( int i=0; i<10; i++ )
    list.add( new Integer( i ));
  else
   for( int i=0; i<10; i++ )
    list.add( null );
 }
}
```

Figure 2. Example program with explicit dynamic loading.

the variable may reference an object reachable from a static field. It guarantees that a node not marked *shared* represents thread-local objects. Such objects cannot be accessed by multiple threads and are subject to thread-local optimizations.

Consider the example in Figure 2. As input to the small program, the user specifies a class that implements the List interface as well as the type of an element (either integer or null). The program instantiates the specified class and repeatedly inserts elements of the specified type into the container. One may write such a program in order to compare the efficiency of various List data structures.

An intraprocedural phase analyzes each method by performing a data-flow analysis on the stack-based bytecode, unifying corresponding nodes at control-flow merges. The intraprocedural analysis of test reveals a very simple picture of the heap (see Table 1). All of test's variables refer to distinct nodes, and the method does not reveal the structure of these nodes. The nodes labeled list and element statically encapsulate the two formal parameters, the node labeled exception corresponds to the exception object that the method may throw, the "int" node denotes the global String constant "int", and the node labeled integer corresponds to the Integer objects appended to the list. A thick border

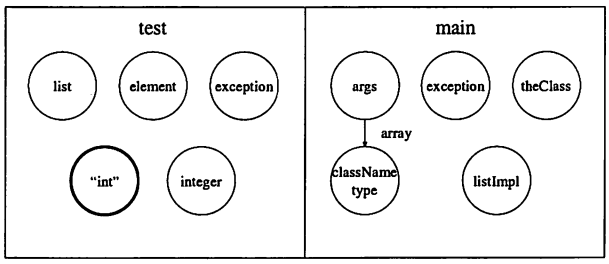


Table 1. Portions of the graph corresponding to the methods test and main.

signals that the "int" node is *shared* and hence may be accessible to multiple threads. Similarly, the intraprocedural analysis of main (also Table 1) reveals five nodes—one for the exception object, one for the listImpl object, one for the Class object, one for the incoming array, and one for the contents of the incoming array. Since a static analysis generally cannot distinguish elements of an array, the variables className and type map to the same node. These subgraphs serve as summaries for the interprocedural portion of the analysis.

The context-sensitive interprocedural phase connects call sites to target methods by mapping the structure of the formal parameters to the corresponding actual parameters. If the caller and callee are in the same strongly connected component (SCC)[1] of the call graph, the analysis merges the node of an actual parameter with the node of the corresponding formal parameter. This unification obviates the need to iterate over the SCC until a fixed point is reached, but it introduces some conservatism. If the caller and callee are in separate SCCs, the analysis imposes the structure of the formal parameters on the actual parameters. However, if the analysis has marked a node in the callee as *shared*, it merges the node with the corresponding node in the caller. This ensures that all methods work with the same *shared* nodes.

The shape analysis is a backward analysis in that it examines a target method before examining a caller method. To accomplish this, it constructs a static call graph and examines each SCC in reverse topological order. Within an SCC, it examines methods arbitrarily.

Without additional information, it is impossible to construct a complete static call graph for our example since the class implementing the List interface is not known statically. This is due to the forName method, which dynamically loads the type of the list, and to the

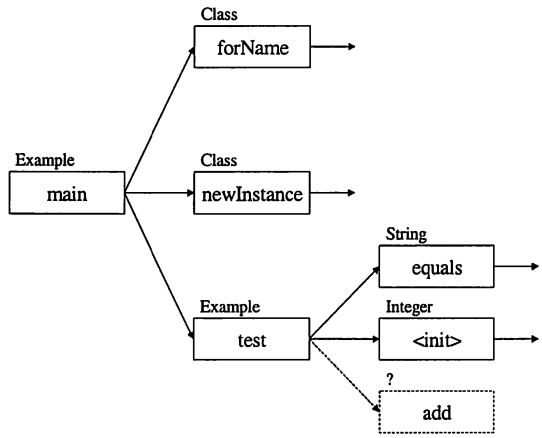


Figure 3. Incomplete call graph of our example.

classpath variable, which gets defined only at run-time. Figure 3 shows an incomplete call graph for our example. The targets of the add invocations cannot be deduced from the program's text. In this case the analysis can give up, can conservatively assume that all objects passed to add become *shared*, or can somehow guess the target methods. The last option may lead to incorrect results.

By performing the analysis at run-time, we can solve the problem of not knowing the call graph because a dynamic analysis can observe the target of the add method. The next section presents an adapted version of the analysis, which can operate dynamically in a JVM.

3. Incremental Shape Analysis

We adapt the above shape analysis to work while the program executes. To be effective, it must work with an incomplete call graph and, as the call graph expands, build upon and modify previous results.

A dynamic analysis avoids the problems caused by dynamic loading and binding because it can observe the targets of call sites. At the same time, this enables it to be more precise than a static analysis for two reasons. First, it knows the exact target(s) of a call site, whereas a static analysis generally amasses a conservative set of potential targets.[2] Second, it only propagates information to call sites that the program executes. For instance, in our example, the dynamic analysis does not need to analyze both calls to add. The next section will explain why this is the case.

[1] A *strongly connected component* is a maximal set of nodes in which there is a path from any node in the set to every other node in the set.

[2] The degree of conservatism depends on the method resolution scheme that the static analysis employs.[14]

```
AnalyzeInvocation( CallSite s, Method t )—
 // step 1
 if <s,t> already analyzed, return
 if t has not been analyzed, analyze t intraprocedurally

 // step 2
 if caller and t are currently in the same SCC
   unify actuals of s and formals of t
 else
   record <s, t>
   DetermineChangesToSCCs( s, t )

 // step 3
 if caller and t are in the same SCC
   PropagateChangesUpCallGraph( t )
 else
   PropagateChangesAlongBinding( s, t )

DetermineChangesToSCCs( CallSite s, Method t )—
 // collapse cycles in call graph involving binding <s,t>
 if RecursivePathExists( method containing s, t, set )
   unify actuals of s and formals of t
   add SCC of t to set

 union all SCCs in set

RecursivePathExists( Method m, Method t, Set set )—
 // recursively collapse cycles in call graph
 if m equals t return true

 sameSCC = false
 for each binding <s,g> such that g and m are in the same SCC
   if RecursivePathExists( method containing s, t, set )
     unify actuals of s and formals of g
     add SCC of m to set
     sameSCC = true

 return sameSCC

PropagateChangesUpCallGraph( Method t )—
 // push changes to all call sites targeting t's SCC
 for each binding <s,g> such that g and t are in the same SCC
   propagate from g to s
   if change occurred to formals of method containing s
     add method containing s to set

 for each m in set
   propagateChangesUpCallGraph( m )

PropagateChangesAlongBinding( CallSite s, Method t )—
 // push summary information from t to s
 propagate from t to s
 if change occurred to formals of method containing s
   propagateChangesUpCallGraph( method containing s )
```

Figure 4. Incremental shape analysis algorithm.

3.1 General Approach

The general algorithm for the incremental analysis appears in Figure 4. Because it is incremental, it does not know the entire call graph at the time of analysis. It consequently works with what it does know and modifies the results as the call graph grows.

The binding of a method to a call site drives the analysis. When a new binding occurs, it performs three steps (see AnalyzeInvocation in Figure 4). First, if it has not already analyzed the target method intraprocedurally, it does so at this time. Second, it identifies any changes to the SCCs as a result of this binding. Third, it propagates the summary of the target method up the call graph. We describe each of these steps in more detail below.

Just as it does in the whole-program version, the analysis first analyzes a method intraprocedurally. The structure of the nodes of the formal parameters will serve as a method summary when the analysis propagates information across call sites. In general, we must maintain information regarding nodes not reachable from the nodes of formal parameters since they may become reachable at a later time. The cost of analyzing a method is a one-time cost; the analysis never needs to analyze it again. Therefore, this cost resembles the cost of bytecode verification and will be nearly negligible for most applications.

To conveniently find the callers of a target method that are outside the target's SCC, the analysis maintains an abbreviated call graph, in which edges between methods within the same SCC are omitted. As the analysis adds edges to this call graph, the SCCs may change. Starting at the new target method, the analysis does a reverse depth-first traversal of the abbreviated call graph. If it

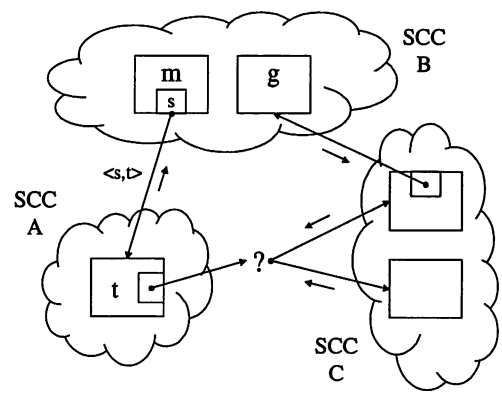


Figure 5. The detection of cycles in the call graph involving the new binding <s,t> (DetermineChangesToSCCs).

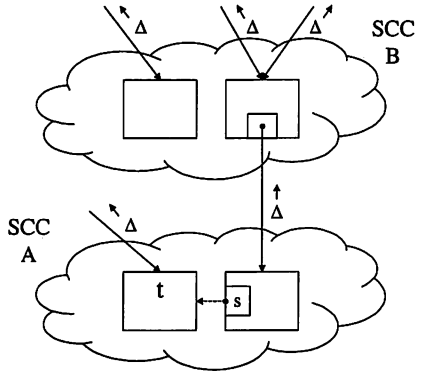


Figure 6. The recursive propagation of summary information along all edges entering t's SCC (PropagateChangesUpCallGraph).

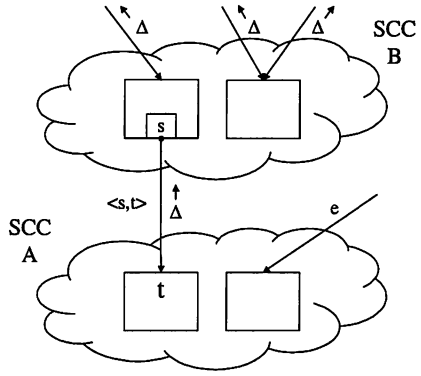


Figure 7. The recursive propagation of summary information starting with the binding <s,t> (PropagateChangesAlongBinding).

returns to the target method, indicating that recursion may occur, it unions all SCCs of the methods along this path and collapses the involved call sites. Figure 5 attempts to illustrate this step. Beginning with the edge <s,t>, the short arrows indicate the traversal of edges between SCCs. Depending on the connectivity of the abbreviated call graph, this may be an expensive operation. In the worst case, the analysis inspects every edge.

The third step propagates the summary of the target method up the call graph. If the caller and the callee are in the same SCC, the analysis must propagate along all call sites that invoke any method in the target's SCC; otherwise, it only needs to propagate along the new binding. Figure 6 depicts the former case, and Figure 7 the latter. Note that the analysis does not propagate along edge e since no changes can occur within t's SCC as a result of the new binding. In either case, the caller then acts as the callee, and its summary flows to its callers. This process continues until either the analysis reaches a root method of the call graph or the structure of the formal parameters of the method containing the call site does not change. In the worst case, the number of propagations is equal to the number of bindings, although we have observed that the test for a change in the formals greatly reduces the number of propagations. Nonetheless, the number of propagations can be several times the number of bindings, as we will see in the next section.

Consider our example in Figure 2 and suppose the program instantiates the class java.util.Vector and adds null references to the list. Figure 8 shows the order in which the incremental analysis constructs and examines the call graph. For clarity, we ignore methods not shown in the diagram. Each step is numbered, where a number inside a box denotes an intraprocedural analysis and a number outside a box indicates a propagation to a call site.

The analysis starts by analyzing main. When the program invokes the forName method, the analysis examines forName in class Class and propagates the information to the call site in main. Next, when the program invokes newInstance, the analysis examines the method and propagates its summary to main. The method newInstance, in turn, calls the constructor of class Vector, <init>. The analysis intraprocedurally analyzes <init>, propagates the summary of <init> to newInstance, and propagates the changes in newInstance to main. The process continues until no changes to the call graph occur.

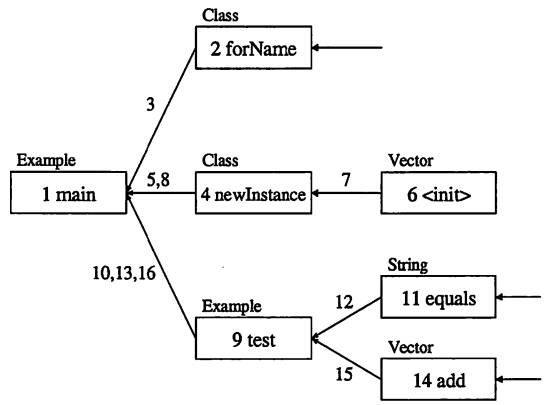


Figure 8. Steps taken by the incremental analysis on our example.

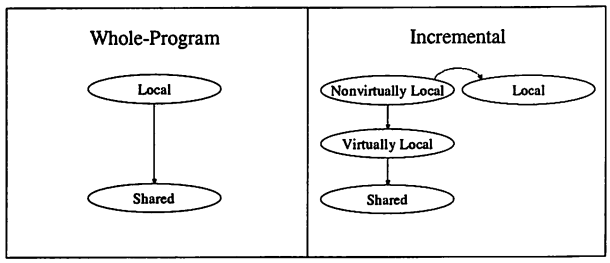


Table 2. Lattices describing the hierarchy of object classifications for the whole-program and incremental approaches.

Since the program never takes the first path of the if-test, the analysis never analyzes the Integer constructor and never propagates information to the first call to add. An off-line whole-program analysis would conservatively do both because it cannot predict the run-time flow of control.

3.2 Classification of Objects

The whole-program version characterizes a node as either *local* or *shared*. A node is initially *local* and becomes *shared* if the analysis merges it with a *shared* node or if it becomes reachable from a *shared* node (see the first column of Table 2). The incremental version introduces two more classifications in order to determine if an object is *guaranteed* to be thread-local, regardless of future classes loaded into the system. We describe all classifications below.

Local—A node is *local* if, no matter what path the program takes and what new classes the system loads, it can never be reachable from a *shared* node.

Nonvirtually Local—A node is *nonvirtually local* if it is not reachable from a node of a variable that is passed into or returned from a virtual method (*i.e.*, it flows into no virtual methods). This node is currently thread-local but may become shared as new methods execute.

Virtually Local—A node is *virtually local* if it flows into a virtual method. In general, a *virtually local* node is not guaranteed to be thread-local because the target of a virtual call site may change in future executions. Therefore, the analysis never knows if it has seen the entire scope of the node.

Shared—A node is *shared* if it is reachable from a *shared* node. The analysis initially marks a node associated with a static field as *shared*.

The relationship of these classifications can be viewed as the second lattice in Table 2. Initially a node is *nonvirtually local*. If it flows into a virtual method, it becomes *virtually local*. If it is stored into or read from a static field, it becomes *shared*. The unification of two nodes takes the meet of the classifications, as defined by the lattice. Additionally, the analysis upholds the rule that a node is at least the lowest type of any of its parents in the resulting graph. Therefore, any node reachable from a *shared* node must also be *shared*, and no node reachable from a *virtually local* node may be *nonvirtually local*.

The classification of *local* is a special case. A node changes from *nonvirtually local* to *local* if, for every method to which the node flows, the method's corresponding node is *local*. This guarantees that *local* nodes can be optimized without later being deoptimized. In our implementation, each *nonvirtually local* node maintains a list of nonvirtual methods to which it flows. After analyzing the binding to one of these methods, we remove the method from the list if the corresponding node in the target is marked *local*. When no methods remain in the list, we promote the node to *local*.

In our example, the node associated with element is *virtually local* because it flows (as the receiver) into the virtual method equals. Similarly, the node associated with variable list is *virtually local* because it flows into the virtual method add. During the initial propagation from test to main, the analysis marks the node associated with listImpl as *virtually local*.

4. Evaluation

An on-the-fly analysis will only be worthwhile if the speed-up resulting from optimizations offsets the time required to carry out the analysis. We assume that an optimization—such as synchronization elimination or stack allocation—depends on the identification of a thread-local object. Thus, in general, the more thread-local objects the analysis identifies, the better.

The optimizer's strategy greatly influences the number of these objects. We label an optimization *pessimistic* if it optimizes an object only when the analysis guarantees that multiple threads cannot access the object. This corresponds to optimizing only objects represented by nodes marked *local*.

In contrast, we label an optimization *optimistic* if it assumes that an object is thread-local before the analysis has analyzed the entire scope of the object. This corresponds to optimizing objects represented by nodes marked *local*, *nonvirtually local*, or *virtually local*.

This distinction affects not only the number of optimizations but also the number of deoptimizations. Consider the following code:

```
x = new
lock x
foo( x )
unlock x


static void foo( p )
{
  X.global = p;
}
```

Suppose foo has not been analyzed at the time of the allocation of x. An optimistic approach immediately optimizes the new object and probably removes the subsequent lock operation. However, after analyzing foo, which makes the new object escape the thread, the optimizer must undo the optimization of x before the body of foo executes. On the other hand, a pessimistic approach waits until it sees the entire scope of x. In this case, it does not optimize x because it has not seen the method foo at the time of the allocation. If foo were not to make x visible to other threads, the pessimistic approach would miss out on the optimization.

The number of optimizable objects also depends on when the dynamic analysis begins. If it starts when the program starts, the optimizer has the potential to capture all optimizable objects. On the other hand, if it starts in the middle of program execution, the optimizer may miss some optimizations. The overall cost of the analysis is smaller the later the analysis begins, thereby encouraging the run-time system to postpone commencement.

We investigate this trade-off by evaluating three approaches. The first begins the analysis immediately in order to capture all optimizable objects. The second delays the analysis a predetermined amount of time, in the hope of reducing the overhead. Finally, the third approach reuses the propagation results of previous analyses to counter the high run-time cost without sacrificing optimizable objects. Before we elaborate on these approaches, we describe our experimental framework and benchmark applications.

4.1 Experimental Framework

We implemented an incremental shape analysis in Java. To allow the analysis to operate on an executing application, we instrument the application to invoke the analysis before key points in its execution. These points are method entry, method exit, call sites, allocation sites, and monitorenter instructions.[1] This instrumentation enables us to identify previously unseen call site/target pairs and to monitor object allocation. We do not trace the execution of system threads; hence all measurements ignore start-up. We ran all tests on a 400MHz Pentium II, using Sun's Java 2 (build 1.3.0-C) for Windows.

Since the JVM disallows both instrumented and uninstrumented versions of core JDK classes, the analysis code is also instrumented. This makes it impossible to determine the running time of our implementation. Notwithstanding, by counting the propagations, we can still sense the overhead of the analysis.

4.2 Benchmarks

Our benchmark suite consists of *jess*, *db*, and *mtrt* from the SPECjvm98 benchmarks [12]; *JLex*; *java_cup*; *slice*; and *volano*. For completeness we include our example program in the first two tables. The multi-threaded *slice* applet, obtained from [10], visualizes radiology data. *Volano*, a multi-threaded chat room simulation, is the client side of VolanoMark 2.1.2.[15]

Table 3 lists the executions we used in our experiment. Some results varied slightly from one run to another, depending on the behavior of the threads. We ran *JLex* on *sample.lex*, which was included in its distribution, and on a homework solution for a compilers class. Similarly, we ran *java_cup* on Java 1.1's grammar, which came with its distribution, and on a homework solution.[2] Due to the significant slowdown caused by our instrumentation, we ran the SPECjvm98 applications with the smallest size input (-s1).

The third column in the table is the number of method invocations during execution of the instrumented program and is indicative of the running time. The number of distinct methods executed, the fourth column, ranges from 18 for our small example to over 1400 for *slice*. This number equals the number of intraprocedural analyses needed. The column "# SCCs" lists the counts of strongly connected components, which are close to the figures of the previous column. This means that few call chains form recursive paths and suggests that the unification of call sites may not lose much, if any, precision. The second-to-last column

[1] The bytecodes invokevirtual, invokespecial, invokeinterface, and invokestatic call methods, and new, newarray, anewarray, and multinewarray allocate heap objects.

[2] The homework input files are available on the first author's homepage (http://www.cs.ucsb.edu/~bogda).

Benchmark	Description	# Method Invocations	# Methods	# SCCs	# Call Site/ Target Pairs	# Lock Operations
db	Database application.	85277	449	439	1744	22141
Example Vector int	Example in paper.	63	18	18	17	10
java_cup on hwk	Parser generator.	551768	761	761	2948	57774
java_cup on java11	Parser generator.	9353838	753	753	2885	574686
jess	Expert system.	599477	850	830	2900	86947
JLex on hwk	Lexical scanner generator.	9191528	243	240	1115	2271197
JLex on sample	Lexical scanner generator.	3807044	242	239	1104	1839304
mtrt	Two-threaded ray tracer.	5721456	582	572	2712	350574
slice	Radiology data viewer.	1847615	1468	1468	3643	26395
volano	Chat room simulation.	9394365	433	433	841	5021842

Table 3. Characteristics of ten executions.

is the number of bindings that trigger the interprocedural propagations. We include the last column, which gives the number of lock acquisitions of each program,[1] to suggest the effectiveness of synchronization elimination. We found no straightforward measure of the effectiveness of stack allocation, other than the count of optimizable objects.

Ruf demonstrated that a number of applications can be analyzed off-line in a matter of seconds.[10] For example, his analysis of *java_cup* finished in 1.01 seconds and *JLex* in 0.56 seconds. Extremely large applications took about twenty seconds to inspect. An on-line analysis has the advantages of seeing a more precise call graph but has the disadvantage of being incremental.

4.3 Immediate Propagations

The first strategy we discuss is one that starts the analysis when the program starts and analyzes all call site/target bindings immediately. By analyzing the target method before it executes, the analysis has the potential to optimize all objects created by the program. Before an allocation site executes, the analysis will have already analyzed the context of the instruction. Once the run-time system has performed an optimization, the analysis must continue to analyze new targets immediately; otherwise, an unanalyzed binding may cause optimized code to execute incorrectly.

The ability to catch new call site targets is straightforward. If the JVM interprets the call site, it triggers the analysis when the target is previously unseen. If native code is executing the call site, the code triggers the analysis after determining the target but before branching to it. To avoid triggering the analysis on every invocation, one may use a polymorphic inline cache and move the triggering to the fallback of the conditional, as follows:

```
if target is A
  jump directly to A
else if target is B
  jump directly to B
else
  execute method lookup code
  trigger the analysis
  jump directly to correct method
```

This example, which assumes that the analysis has already seen targets A and B, activates the analysis only when the uncommon target arises.

Table 4 illustrates the cost and effectiveness of starting the interprocedural analysis immediately. The second column gives the number of times the analysis propagates a method's summary to a call site. It does not include the unification of arguments for methods within an SCC. This number, which ranges between 2.0 and 3.3 times greater than the number of distinct call site/target bindings, ultimately governs the overhead of the analysis.

The next five columns reveal the types of objects allocated. To determine the type, we look up the classification of the node corresponding to the allocation site and find the most recent method on the call stack in which the corresponding node cannot escape. This allows us to characterize objects that leave factory-type methods with respect to the calling contexts. Of the potential thread-local objects, nearly all objects are classified as *virtually local*. This corresponds to a typical object-oriented program's high use of virtual methods. The most frequently allocated *local* object is the 12-byte array in toString of class Integer. Because of

[1] We did not count entry into synchronized static methods.

Benchmark	# Propa-gations	# Objs. Allocated					# Objs. Optimized		# Objs. Deoptimized		# Lock Ops Eliminated	
		Local	Nonvirt Local	Virt. Local	Shared	Un-known	Opt.	Pes.	Opt.	Pes.	Opt.	Pes.
db	4828	51 (1%)	59 (1%)	2532 (53%)	2133 (45%)	0	2642 (55%)	51 (1%)	901 (34%)	0	12816 (58%)	0
Example	34	0	0	11 (100%)	0	0	11 (100%)	0	0	0	10 (100%)	0
java_cup on java11	9384	32007 (5%)	65 (0%)	127521 (20%)	487226 (75%)	0	159593 (25%)	32007 (5%)	654 (0%)	0	228227 (40%)	0
jess	9667	207 (0%)	404 (1%)	17025 (36%)	19065 (40%)	10914 (23%)	17636 (37%)	207 (0%)	210 (1%)	0	50556 (58%)	0
JLex on sample	3465	440 (1%)	178 (0%)	46165 (97%)	980 (2%)	0	46783 (98%)	440 (1%)	10 (0%)	0	1838356 (100%)	0
mtrt	7326	11038 (4%)	125 (0%)	273561 (91%)	14625 (5%)	0	284724 (95%)	11038 (4%)	100 (0%)	0	349412 (100%)	0
slice	10174	2366 (1%)	578 (0%)	452615 (96%)	13890 (3%)	0	455559 (97%)	2366 (1%)	700 (0%)	0	16626 (63%)	28 (0%)
volano	2269	45727 (5%)	625 (0%)	781332 (93%)	8469 (1%)	0	827684 (99%)	45727 (5%)	71 (0%)	0	5015018 (100%)	0

Table 4. Results of an analysis that begins immediately.

our implementation, we were unable to determine the types of a handful of objects in *jess*.

The remaining columns compare the optimistic and pessimistic optimization approaches, commencing with the number of optimizable objects. For all benchmarks, the number of objects in the pessimistic case is significantly smaller than in the optimistic case. This follows because the analysis does not classify many objects as *local*. Objects tend to flow into virtual methods, and programs do not always execute every call site.

The next comparison is on the number of objects that would need to be deoptimized as a result of marking a node *shared*. The pessimistic approach never needs to deoptimize since, by definition, it does not optimize an object unless it is guaranteed to be thread-local. The optimistic approach, on the other hand, will need to deoptimize, although the figures are much lower than the number of objects optimized. Aside from *db*, it needs to undo between 0% and 1% of its optimizations. *Db* is the exception. Even though the analysis demotes only 27 nodes (1%) to *shared* status, these demotions invalidate 901 optimized objects (34%).

The last comparison is on the number of lock acquisitions that can be removed. The optimistic approach has mixed success; it can remove nearly all lock acquisitions in *JLex* but less than half in *java_cup*. In comparison, the pessimistic approach tends not to optimize objects that are later locked. Only for *slice* does the pessimistic approach remove any locking. Because the pessimistic approach does not prove effective for synchronization elimination on these applications, we disregard it in future tables.

4.4 Delayed Propagations

The efficiency of the static whole-program analysis stems from the fact that it analyzes a binding only once. The incremental version, on the other hand, analyzes a binding repeatedly, as the call graph grows and call sites need updating. This behavior fights the natural progression of program execution.

Nothing requires the analysis to start immediately. To counter the high number of propagations, we can defer the analysis until the program has established a sufficiently large call graph. At this point, the analysis can analyze the SCCs of the current call graph in reverse topological order, thereby reducing the initial number of propagations. After an optimization occurs, however, the analysis must begin an immediate style in order to identify *shared* nodes before the program incorrectly executes.

Since the run-time system may start the analysis at any time, it may be able to hide the analysis behind I/O, time-consuming memory accesses, or garbage collection. The initial interprocedural phase does not

Benchmark	# Meth. Before Prop.	# Propa-gations	# Objs. Allocated					# Objs. Opt.	# Objs. Deopt.	# Lock Ops Elim.
			Local	Nonvirt Local	Virtually Local	Shared	Un-known			
db	430	732 (↓ 85%)	11 (0%)	9 (0%)	446 (9%)	33 (1%)	4276 (90%)	466 (9%)	14 (3%)	4197 (19%)
java_cup on java11	190	8046 (↓ 14%)	31907 (5%)	45 (0%)	117445 (18%)	485906 (75%)	11516 (2%)	149397 (23%)	353 (0%)	127737 (22%)
jess	708	5552 (↓ 43%)	177 (0%)	69 (0%)	10835 (23%)	8406 (18%)	28127 (59%)	11082 (23%)	61 (1%)	25498 (29%)
JLex on sample	193	2103 (↓ 39%)	439 (1%)	17 (0%)	44022 (92%)	963 (2%)	2322 (5%)	44478 (93%)	3 (0%)	1686615 (92%)
mtrt	391	4266 (↓ 42%)	10937 (4%)	34 (0%)	102274 (34%)	12777 (4%)	173327 (58%)	113245 (38%)	43 (0%)	3753 (1%)
slice	404	8660 (↓ 15%)	2344 (1%)	512 (0%)	450976 (96%)	13018 (3%)	2255 (0%)	453832 (97%)	153 (0%)	11606 (44%)
volano	318	1817 (↓ 20%)	54612 (6%)	591 (0%)	779255 (92%)	8058 (1%)	2637 (0%)	825458 (98%)	66 (0%)	5007397 (100%)

Table 5. Results of an analysis that begins after 50,000 instrumentation ticks of no new methods.

need to complete before the program resumes; it may be interspersed.

When should a delayed analysis start? If it waits too long, it will miss chances for optimization. In the worst case, an application allocates all optimizable objects before the analysis begins. If it starts too early, it will face the same number of propagations as the immediate approach. As a compromise, a delayed analysis could start when the rate of class loading slows, when the rate at which new methods execute slows, or when the run-time system detects a frequently executed portion of the application. In any case, one can devise an application that countermines the chosen delay strategy.

We ran an experiment in which the analysis started after 50,000 instrumentation ticks had occurred without causing a new method to be executed. We felt that this number would give a program ample time to settle down. The results appear in Table 5.

The second column of the table lists the number of methods executed before the analysis starts. The number of propagations, listed in the third column, is smaller than in the immediate case, reducing it on average by 37%. In general, however, additions to the call graph at the end of the program often require more propagations than additions at the beginning, causing the bulk of the propagations to remain in the delayed approach.

The distribution of the types of objects allocated is nearly identical to the previous approach. Unknowns result when programs allocate objects before the initial analysis has run. These potentially translate into missed opportunities for optimization. For example, the delayed analysis eliminates about 50% fewer lock acquisitions for *jess*.

4.5 Persistent Propagations

A viable strategy must have, in the common case, a low run-time overhead and a high potential for optimization. We have seen that an immediate analysis has the potential to discover all optimizable objects but may incur high propagation costs. We have also seen that delaying an initial analysis can reduce the number of propagations but may miss optimizations.

We ran a third experiment in which the analysis utilized previous results on a given application. Table 6 shows measurements of two scenarios. The first analyzes *JLex* on *hwk*, using the call graph and results of *JLex* on *sample*. Even though the *hwk* input file is much larger than *sample*, the analysis only analyzes 2 additional methods and 18 additional bindings. This causes 63 propagations—a huge drop from 3465 of the first input. Moreover, the run-time system can optimize a high percentage of objects and does not need to undo any of the optimizations.

The second scenario analyzes *java_cup* on *hwk* using the results of *java_cup* on *java11*. This time *hwk* is a much smaller input than the first input file. The number of previously unseen methods and bindings again decreases significantly, incurring fewer than 200 propagations.

Benchmark	# Meth. Analyzed	# Invoc. Analyzed	# Propagations	# Objs. Allocated				# Objs. Optimized	# Objs. Deoptimized	# Lock Ops Elim.
				Local	Nonvirt. Local	Virtually Local	Shared			
1. JLex on sample	242	1104	3465	440 (1%)	178 (0%)	46165 (97%)	980 (2%)	46783 (98%)	10 (0%)	1838356 (100%)
2. JLex on hwk	2	18	63	2241 (0%)	313 (0%)	472609 (98%)	4713 (1%)	475163 (99%)	0 (0%)	2269005 (100%)
1. java_cup on java11	753	2885	9384	32007 (5%)	65 (0%)	127521 (20%)	487726 (75%)	159593 (25%)	654 (0%)	228227 (40%)
2. java_cup on hwk	8	89	190	4318 (9%)	29 (0%)	10992 (22%)	34402 (69%)	15339 (31%)	1 (0%)	39809 (69%)

Table 6. Results of an analysis that utilizes previous results.

Call sites typically target the same methods on subsequent executions. By reusing results, this approach avoids analyzing a binding repeatedly and skirts deoptimization. If the dynamic properties of an execution change (for example, the input changes or the classpath variable is modified, leading to a different binding at a nonvirtual call site), the analysis will catch the changes and take the union of the previous and new scenarios. The result, although less precise, is still safe.

This approach still has several disadvantages. First, a *shared* node will persist in all subsequent executions even if these executions do not let the node escape the thread. For example, suppose one execution of our Example program uses a List data structure that places itself in a static field. The analysis will mark the list *shared* and on subsequent executions will never consider optimizing the list. Second, the analysis may miss the (possibly rare) situation where it is better to optimize an object and later deoptimize it than to disqualify it from optimization.

The results of Section 4 seem to suggest the following optimization strategy. On the first execution of a program, perform no analysis while the program executes but record the binding of call sites to target methods. This sacrifices optimizations in the current execution but incurs little overhead. After the program finishes, use an efficient whole-program version to analyze the entire program and then save the results. On subsequent executions use the saved results and an optimistic strategy to optimize the program.

5. Related Work

Shape analysis has recently entered the scene to identify optimizable objects in Java. Researchers aim to stack allocate and to remove synchronization on thread-local objects.[1,3,4,5,6,10,16] All of these analyses assume a closed, known world and work in a static compiler. Shape analysis is also being used to help program verification and model checking.[7,9] At this time, however, it is not clear if these latter uses will benefit from a dynamic, incremental approach.

The field analysis by Ghemawat and Randall [8] avoids having to know the entire program by looking at the access flags of fields and methods. For example, members with package scope can be accessed only within the enclosing package. This idea, coupled with the idea of package sealing [18], which restricts all classes within a package to come from the same archive file, may allow us to classify more objects as *local*. For instance, a *virtually local* node may change to *local* if it only flows to sealed call sites and if all potential targets have been analyzed. Our current analysis examines neither access flags nor type information.

The most common dynamic analysis for Java is just-in-time compilation. It trades off compilation overhead for increased execution. Since it is not a whole-program analysis, it can be selective about what it compiles. For example, it may choose to compile only frequently executing methods. If the compilation becomes too expensive, it can fall back to a naïve native code translation or even to interpreted code. A dynamic shape analysis faces a different battle because it cannot simply quit if the analysis becomes too time consuming. If the analysis fails to analyze a method, an incorrect execution may ensue.

Two recent publications move toward dynamic interprocedural analyses. The first, by Sreedhar *et al.* [13], presents a framework called *extant analysis* that, during an off-line static analysis, characterizes all references as either *unconditionally extant* or *conditionally extant*. The former denotes a reference to

Optimization Approach	Benefits	Drawbacks
Optimistic	Has the potential to optimize all objects	May need to deoptimize
Pessimistic	Precludes deoptimization	Guarantees few thread-local objects May miss optimizations

Table 7. Trade-offs regarding the optimization strategy.

Analysis Approach	Benefits	Drawbacks
Immediate	Has the potential to optimize all objects	Faces numerous propagations
Delayed	Reduces the number of propagations	May miss optimizations
Persistent	Infrequently propagates after the first execution	Requires additional start-up and exit costs Inherits worst case over all executions

Table 8. Trade-offs regarding the start of the interprocedural analysis.

an object whose type is guaranteed to be in a specified closed world, and the latter captures the remaining references. In our work, our notion of a closed world gets defined as the program executes. If a reference flows into a virtual method (*i.e.*, its node is *virtually local*), it exits the closed world. Only fully analyzed nodes that remain in the closed world can be guaranteed to be thread-local.

The second, by Serrano *et al.* [11], introduces a quasi-static analysis, named Quicksilver, that saves compiled code between program executions. Before execution, it validates the existing code images and "stitches" them to reflect the current run-time properties. Their approach is effective for the SPECjvm98 benchmarks, once it has processed an initial "learning" execution. This approach is similar to our persistent strategy, although we do not require stitching; we take the union over all previous executions.

6. Conclusions

We adapted an efficient whole-program shape analysis to operate incrementally and on-the-fly. Its dynamic property enables it to observe the dynamic call graph, to analyze only the executing methods, and to propagate information only to executing call sites. Also, the dynamic behavior allows it to perform optimizations not expressible in bytecode.

In general, when a call site targets a new method, the shape analysis propagates information from the target method to all affected parts of the call graph. Table 7 and Table 8 summarize various approaches to performing the analysis and a related optimization. Our results suggest that an optimizer must optimistically select objects to optimize in order to identify a large number of optimization points. For the applications we studied, an optimizer that takes this approach will need to undo only a small fraction of its optimizations. A strategy that propagates immediately is able to optimize objects as soon as possible but incurs a large cost for propagations. To alleviate the propagation cost, we can delay the initial analysis. Doing so, however, does not significantly reduce the number of propagations. We can eliminate most of the propagations if we reuse the analysis results from previous executions.

This paper does not attempt to close the book on the dynamic shape analysis problem; it provides some empirical evidence of the difficulty of performing it dynamically and suggests an approach that may be both effective and attainable. Two related issues must be resolved before it can be used in practice.

First, because any thread of the program may trigger the analysis, it must be thread-safe. Moreover, it must handle concurrent activation without sacrificing efficiency. If the analysis uses union-find data structures to efficiently merge nodes and SCCs, as suggested in [10], work done by Anderson and Woll regarding parallel union-find algorithms [2] may prove helpful.

Second, optimization and deoptimization techniques must be explored. Deoptimization is especially difficult. When the analysis marks a node *shared*, it must deoptimize all objects that are represented by this node. This requires the run-time system to identify all objects allocated as a result of the optimization and possibly to recompile optimized code. In the case of lock elimination, the process not only needs to enable all future lock operations on the deoptimized object but also needs to grant the owning thread the current number of nested lock acquisitions that would have been held had the object not been optimized. In the case of stack allocation, it may need to move an optimized object from the stack to the heap.

We hope that this study encourages JVM implementors to consider whole-program analyses that aid in dynamic optimizations. We are currently looking into a dynamic shape analysis that can be driven by frequently accessed objects instead of by changes to the call graph. Such an approach may be able to eliminate the requirement that the analysis examine the entire program.

Acknowledgments

This work was funded in part by the NSF grant CCR-9972571.

References

[1] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In *Proceedings of the Sixth International Static Analysis Symposium*, Venezia, Italy, September 1999.

[2] Richard J. Anderson and Heather Woll. Wait-free Parallel Algorithms for the Union-Find Problem. In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing*, pages 370-380, New Orleans, Louisiana, 5-8 May 1991.

[3] Jeff Bogda. Detecting Read-Only Methods in Java. In *Proceedings of the Fifth International Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR '00)*, pages 143-154, Rochester, New York, 25-27 May 2000.

[4] Jeff Bogda and Ambuj Singh. *Critical Section, Be Gone!* Technical Report TRCS00-18, Department of Computer Science, University of California, Santa Barbara, August 2000.

[5] Jeff Bogda and Urs Hölzle. Removing Unnecessary Synchronization in Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 35-46, Denver, Colorado, 1-5 November 1999.

[6] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape Analysis for Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 1-19, Denver, Colorado, 1-5 November 1999.

[7] James C. Corbett. *Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs.* Technical Report ICS-TR-98-20, Department of Information and Computer Science, University of Hawaii, 14 October 1998.

[8] Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. Field Analysis: Getting Useful and Low-Cost Interprocedural Information. In *Proceedings of Programming Languages, Design, and Implementation (PLDI '00)*, pages 334-344, Vancouver, Canada, 18-21 June 2000.

[9] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting Static Analysis to Work for Verification: A Case Study. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '00)*, pages 26-38, Portland, Oregon, 21-24 August 2000.

[10] Erik Ruf. Effective Synchronization Removal for Java. In *Proceedings of Programming Languages, Design, and Implementation (PLDI '00)*, pages 208-218, Vancouver, Canada, 18-21 June 2000.

[11] Mauricio Serrano, Rajesh Bordawekar, Sam Midkiff, and Manish Gupta. Quicksilver: A Quasi-Static Compiler for Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 66-82, Minneapolis, Minnesota, 15-19 October 2000.

[12] SPEC Java virtual machine benchmark suite. Standard Performance Evaluation Corporation. *SPECjvm98 Documentation, Release 1.0.* August 1998. http://www.spec.org/osq/jvm98/jvm98/doc/index.html.

[13] Vugranam C. Sreedhar, Michael Burke, and Jong-Deok Choi. A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading. In *Proceedings of Programming Languages, Design, and Implementation (PLDI '00)*, pages 196-207, Vancouver, Canada, 18-21 June 2000.

[14] Frank Tip and Jens Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 281-293, Minneapolis, Minnesota, 15-19 October 2000.

[15] Volano benchmark application. http://www.volano.com/benchmarks.html.

[16] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 187-206, Denver, Colorado, 1-5 November 1999.

[17] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape Analysis. In *Proceedings of Conference on Compiler Construction*, Berlin, Germany, 27 March - 2 April 2000.

[18] Ayal Zaks, Vitaly Feldman, and Nava Aizikowitz. Sealed Calls in Java Packages. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 83-92, Minneapolis, Minnesota, 15-19 October 2000.

SableVM: A Research Framework for the Efficient Execution of Java Bytecode*

Etienne M. Gagnon and Laurie J. Hendren
Sable Research Group
School of Computer Science
McGill University
[gagnon,hendren]@sable.mcgill.ca

Abstract

SableVM is an open-source virtual machine for Java intended as a research framework for efficient execution of Java bytecode[1]. The framework is essentially composed of an extensible bytecode interpreter using state-of-the-art and innovative techniques. Written in the C programming language, and assuming minimal system dependencies, the interpreter emphasizes high-level techniques to support efficient execution.

In particular, we introduce a *biderectional layout* for object instances that groups reference fields sequentially to allow efficient garbage collection. We also introduce a *sparse interface virtual table layout* that reduces the cost of interface method calls to that of normal virtual calls. Finally, we present a technique to improve thin locks[13] by eliminating busy-wait in presence of contention.

1 Introduction & Motivation

Over the last few years, Java[21] has rapidly become one of the most popular general purpose object-oriented (OO) programming languages. Java programs are compiled into *class files* which include type information and platform independent bytecode instructions. On a specific platform, a runtime system called a *virtual machine*[24] loads and links class files then executes bytecode instructions. The virtual machine collaborates with the standard class libraries to provide key services to Java programs, including threads and synchronization, automatic memory management (garbage collection), safety features (array bound checks, null pointer detection, code verification), reflection, dynamic class loading, and more.[2]

Early Java virtual machines were simple bytecode interpreters. Soon, the quest for efficiency led to the addition of *Just-In-Time compilers* (JIT) to virtual machines, an idea formerly developed for other OO runtime systems like Smalltalk-80[17] and Self-91[15]. In a few words, a just-in-time compiler works by compiling bytecodes to machine specific code on the first invocation of a method. JITs range from the very naive, that use templates to replace each bytecode with a fixed sequence of native code instructions (early versions of Kaffe[5] did this), to the very sophisticated that perform register allocation, instruction scheduling and other scalar optimizations (e.g. [8, 23, 29, 32]).

JITs face two major problems. First, they strive to generate good code in very little time, as compile time is lost to the running application. Second, the code of compiled method resides in memory; this augments the pressure on the memory manager and garbage collector. Recent virtual machines try to overcome these problems. The main trend is to use dynamic strategies to find *hot* execution paths, and only optimize these areas (e.g. [4, 10, 16]). HotSpot[4], for example, is a mixed interpreter and compiler environment. It only compiles and optimizes *hot spots*. Jalapeno[10, 11], on the other hand, always compiles methods (naively at first),

*This research is partly supported by FCAR, NSERC, and Hydro-Québec.

[1] In this document, the term *Java* means: *the Java programming language.*

[2] There exist static compilers that directly compile Java programs to machine code (e.g. [2, 3, 7]). The constraints of static and dynamic environments are quite different. Our research focuses on dynamic Java execution environments.

then uses adaptive online feedback to recompile and optimize hot methods. These techniques are particularly suited to virtual machines executing long running programs in server environments. The optimizer can be relatively slow and consist of a fully fledged optimizing compiler using intermediate representations and performing *costly* aggressive optimizations, as compile time will be amortized on the overall running time.

Our research complements these approaches by exploring opportunities for making the virtual machine execute efficiently. Rather than looking at fine grain techniques, like register allocation and instruction scheduling, we address the fundamental problem of data layout in a dynamic Java environment. While Java shares many properties with other object-oriented languages, the set of runtime constraints enforced by the verifier and the basic services provided for each object (hash code, locking) are unique. This leads us to revisit traditional data structures used in object-oriented runtime environments, and adapt them to fully take advantage of the properties of the Java runtime environment.

As a testbed for evaluating our proposed data structures and algorithms, we are designing and implementing SableVM, a standards conforming open-source virtual machine. Written in the C programming language, and depending on the POSIX application programming interface (API), it is meant as a small and portable interpreter[3]. It can be used as an experimental framework for extending the bytecode language. It can also be used as an efficient virtual machine for embedded systems, or as a profiling interpreter in a hybrid interpreter/just-in-time optimizing-compiler environment.

The remaining part of this document is structured as follows. Section 2, we state the contributions of this paper. In section 3, we give an overview of the SableVM framework. In section 4, we describe SableVM's threaded interpreter. In section 5, we introduce our classification of virtual machine memory. In section 6, we introduce our new layouts for object instances and virtual tables, and our improved thin locks. In section 7, we discuss our proposed experiments. Finally, in section 8, we present our conclusions.

[3]SableVM depends on the open-source *GNU Classpath[1]* class library for providing standard library services.

2 Contribution

The specific contributions of this paper are as follows.

- Introduction of a bidirectional object instance layout that groups reference fields sequentially, enabling simpler and faster garbage collection tracing.
- Introduction of a sparse interface virtual table layout that enables constant time interface method lookup in presence of dynamic loading.
- Improvement of the bimodal field thin lock algorithm[13, 26] to eliminate busy-wait, without overhead in the object instance layout.
- Categorization of virtual machine memory into separate conceptual areas exhibiting different management needs.

3 Framework Overview

As shown in Figure 1, the SableVM experimental framework is a virtual machine composed of five main components: interpreter, memory manager, verifier, class loader, and native interface. In addition, the virtual machine implements various services required by the class library (e.g.: synchronization and threads).

SableVM is entirely[4] written in portable C. Thus, its source code is readable and simple to modify. This makes an ideal framework for testing new high-level implementation features or bytecode language extensions. For example, adding a new arithmetic bytecode instruction entails making a minor modification to the class loader, adding a few rules to the verifier, and finally adding the related interpreter code. This is pretty easy to do in SableVM, as compared to a virtual machine written in assembly language, or a virtual machine with an embedded compiler (e.g. JIT).

The current implementation of SableVM targets the Linux operating System on Intel x86 processors. It

[4]Exceptions: We assume a POSIX system library, we use *label as values* (see Figure 2(b)), and there is a single line of assembly code (*compare-and-swap*).

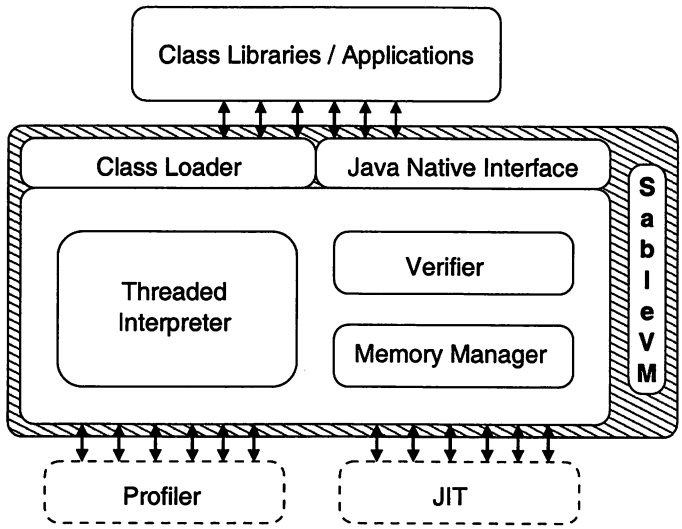


Figure 1: The SableVM experimental framework

uses the GNU libc implementation of POSIX threads to provide preemptive operating system level threads.

4 Threaded Interpreter

SableVM's interpreter is a threaded interpreter. Pure bytecode interpreters suffer from expensive dispatch costs: on every iteration, the dispatch loop fetches the next bytecode, looks up the associated implementation address in a table (explicitly, or through a *switch* statement), then transfers the control to that address. *Direct threading*[20] reduces this overhead: in the executable code stream, each bytecode is replaced by the address of its associated implementation. In addition, each bytecode implementation ends with the code required to dispatch the next opcode. This is illustrated in figure 2. This technique eliminates the table lookup and the central dispatch loop (thus eliminating a branch instruction to the head of the loop). As these operations are expensive on modern processors, this technique has been shown to be quite effective[20, 27].

Method bodies are translated to threaded code on their first invocation. We take advantage of this translation to do some optimizations. For example, we precompute absolute branch destinations, we translate overloaded bytecodes like the `GET_FIELD` instruction to separate implementation addresses (`GET_FIELD_INT`, `GET_FIELD_FLOAT`, ...), and we inline *constant pool* references to direct operand values.

This one pass translation is much simpler than the translation done by even the most naive just-in-time compiler, as each bytecode maps to an address, not a variable sized implementation. However, unlike a JIT, the threaded interpreter still pays the cost of an instruction dispatch for each bytecode. Piumarta[27] has shown a technique to eliminate this overhead within a *basic block* using selective inlining in a portable manner, at the cost of additional memory[5]. SableVM implements this technique optionally through a compile-time flag, as it might not be appropriate for systems with little memory.

5 Memory Management

Memory management is a central issue in the design of SableVM. Most of the high-level performance enhancements introduced in this research are related to memory management.

In this section, we classify the memory of the Java virtual machine according to the control on its management, and its allocation and release behavior. We define four categories (system, shared, thread specific, and class loader specific), and discuss how SableVM takes advantage of them.

[5] On some processors, this technique requires one line of assembly code to synchronize the instruction and data caches.

```
/* code */

char code[] = {
  ICONST_2, ICONST_2,
  ICONST_1, IADD, ...
}
char *pc = code;

/* dispatch loop */

while(true) {
  switch(*pc++) {
  case ICONST_1: *++sp = 1; break;
  case ICONST_2: *++sp = 2; break;
  case IADD:
    sp[-1] += *sp; --sp; break;
  ...
}}
```

(a) Pure bytecode interpreter

```
/* code */

void *code[] = {
  &&ICONST_2, &&ICONST_2,
  &&ICONST_1, &&IADD, ...
}
void **pc = code;

/* implementations */

goto **(pc++);

ICONST_1: *++sp = 1; goto **(pc++);
ICONST_2: *++sp = 2; goto **(pc++);
IADD:
  sp[-1] += *sp; --sp; goto **(pc++);
...
```

(b) Threaded Interpreter

Figure 2: Pure and threaded interpreters

5.1 System Memory

System memory is the portion of memory on which we, as C developers, have essentially no direct control. It consists of the memory used to store executable machine code, native C stacks, the C heap (`malloc()` and `free()`), dynamically linked native libraries, and any other uncontrollable memory.

5.2 Shared Memory

Shared memory is managed by the virtual machine and potentially allocated and modified by many threads executing methods of various class loaders.

This memory consists primarily of the Java heap (which is garbage collected), and global JNI references. The allocation and release behavior of such memory is highly application dependent, with no general allocation or release pattern.

5.3 Thread Specific Memory

Thread specific memory is also managed by the virtual machine, but it is allocated specifically for internal management of each Java thread.

This memory consists primarily of Java stacks, JNI local reference frames for each stack, and internal structures storing thread specific data like stack information, JNI virtual table, and exception status.

This memory exhibits precise allocation and release patterns. Thread specific structures have a life time similar to their related thread. So, this memory can be allocated and freed (or recycled) at the time of respective creation and death of the underlying thread. Also, stacks have a regular pattern: they grow and shrink on one side only. This property is shared by JNI local reference frames.

5.4 Class Loader Specific Memory

Class loader specific memory is managed by the virtual machine and is allocated for internal management of each class loader.

This memory consists primarily of the internal data structures used to store class loader, class, and method data structures. This includes method bodies in their various forms like bytecode, direct threaded code, inlined threaded code, and potentially compiled code (in the presence of a JIT). It also includes normal and interface virtual tables.

This memory exhibits precise allocation and release patterns. This memory is allocated at class loading time, and at various preparation, verification, and resolution execution points. This memory differs significantly from stacks and the shared garbage collected heap in that once it is allocated, it must stay at a fixed location, and it is unlikely to be released soon. The Java virtual machine specification allows for potential unloading of all classes of a class loader as a group, if no direct or indirect references to the class loader, its classes, and theirs instances

remain. In such a case, and if a virtual machine supports class unloading, all memory used by a class loader and its classes can be released at once.

5.5 SableVM Implementation

In SableVM, thread specific memory is managed independently from shared memory. SableVM allocates thread structures on thread creation but does not release them at thread death. Instead, it manages a free list to recycle this memory on future thread creation.

Java stacks are growing structures; a memory block is allocated at thread creation, and if later the stack proves too small, the memory block is expanded, possibly moving it to another location to keep the stack contiguous and avoid fragmentation.

SableVM also manages class loader specific memory independently from other memory. Each class loader has it own memory manager that allocates memory (from the system) in relatively big chunks, then redistributes this memory is smaller fragments. This has many advantages.

It allows the allocation of many small memory blocks without the usual memory space overhead, as `malloc()` would use additional memory space to store the size of each allocated block in prevision of future `free()` calls. In the class loader specific memory category, smaller fragments will only be returned to the system as a group (in case of class unloading), so we need not keep track of individual fragment sizes.

As a corollary, class unloading is more efficient using a dedicated memory manager than using regular `malloc()` and `free()` calls, as there is no need to incrementally aggregate small memory segments, as would happen with a sequence of `free()` calls.

Using dedicated memory managers allows class parsing and decoding in one pass without memory overhead, by allocating many small memory blocks. This is usually not feasible, as it is not possible to estimate the memory requirement for storing internal class information before the end of the first pass.

Finally, and importantly, a dedicated memory manager allows for irregular memory management strategies: it is possible to return sub-areas of an allocated block to the memory manager, if these sub-areas are known not to be used. We take advantage of this to improve the representation of interface method lookup tables[6].

6 Performance Enhancements

In this section, we introduce new layouts for object instances and interface virtual tables, as well as improvements to the thin lock algorithm, leading to high-level performance enhancements in the areas of garbage collection, interface method invocation, and synchronization.

We say *high-level* enhancements, because these techniques are applicable to any Java virtual machine, independently from its form: interpreter, just-in-time compiler, adaptive online feedback based systems, etc.

6.1 Bidirectional Object Layout

In this subsection, we propose a new object layout that optimizes the placement of reference fields to allow efficient gc tracing.

The Java heap is by definition a garbage collected area. A Java programmer has no control on the deallocation of an object. Garbage collectors can be divided into two major classes: tracing and non-tracing collectors. Non-tracing collectors (mainly *reference counting*) cannot reclaim cyclic data structures, are a poor fit for concurrent programming models, and have a high reference count maintenance overhead. For this reason, Java virtual machine designers usually opt for a tracing collector.

There exist many tracing collectors[22]. The simplest models are mark-and-sweep, copying, and mark-compact. The common point to all tracing collectors (including advanced generational, conservative and incremental techniques) is that they must trace a subset of the heap, starting from a root set, looking for *reachable* objects. Tracing is often one of the most expensive steps of garbage collection[22]. For every root, the *garbage collector* (gc) looks up the type of the object to find the offset of its reference fields, then it recursively visits the objects referenced by these fields.

[6] See section 6.2.

To provide efficient field access, it is desirable to place fields at a constant offset from the object header, regardless of inheritance. This is easily achieved in Java as instance fields can only be declared in classes (not in interfaces), and classes are restricted to single inheritance. Fields are laid out consecutively after the object header, starting with super class fields then subclass fields, as shown in Figure 3(a). When tracing such an object, the garbage collector must access the object's class information to discover the offset of its reference fields, then access the superclass information to obtain the offset of its reference fields, and so on. As this process must be repeated for each traced object, it is quite expensive.

There are three improvements that are usually applied to this naive representation. Firstly, reference fields are grouped together in the layout of each class. Secondly, each class stores an array of offsets and counts of reference fields for itself and all its super classes. Thirdly, a *one word* bit array is used in the virtual table to represent the layout of reference fields in small objects (each bit being set if the object instance word, at the same index, is a reference). This is shown in Figure 3(b). For big objects, the number of memory accesses needed to trace an object is $n + 3 + (2 * arraysize)$, where n is the number of references. Two nested loops (and loop variables) are required: one to traverse the array, and one for each array element (accessing the related number of references). For smaller objects, the gc needs to access the virtual table to retrieve the bit field word, then it needs to perform a set of shift and mask operations to find the offset of reference fields. Overall, using this layout, tracing an object is a relatively complex operation.

Tracing reference fields could be much simpler if they were simply grouped consecutively. The difficulty is to group them while keeping the *constant offset* property in presence of inheritance.

We introduce a *bidirectional object instance layout* that groups reference fields while maintaining the *constant offset* property. The left part of Figure 4 illustrates this new layout. In the bidirectional object instance layout, the instance *starting point* is possibly a reference field. The instance grows both ways from the *object header*, which is located in the middle of the instance. References are placed before the header, and other fields are placed after it. The right part of Figure 4 illustrates the layout of array instances. Array element are placed in front or after the array instance header, depending on whether the element type is a reference or a non-reference type, respectively.

The object header contains two words (three for arrays). The first is a *lock word* and the second is a virtual table pointer. We use a few low-order bits of the lockword encode the following information:

- We set the last (lowest order) bit to one, to differentiate the lock word from the preceding reference fields (which are pointers to aligned objects, thus have their last bit set to zero).
- We use another bit to encode whether the instance is an object or an array.
- If it is an array, we use 4 bits to encode its element type (boolean, byte, short, char, int, long, float, double, or reference).
- If it is an object, we use a few bits to encode (1) the number of references and (2) the number of non-reference field words of the object, (or special overflow values, if the object is too big).

We also use two words of the virtual table (see Figure 5) to encode the number of reference and non-reference field words of the object if the object is too big to encode this information in the header.

At this point, we must distinguish the two ways in which an object instance can be reached by a tracing collector. The first way is through an object reference that points to the object header (which is in the middle of the object). The second way is through its starting point, in the *sweep* phase of a mark-and-sweep gc, or in the *tospace* scanning of a copying gc. In both cases, our bidirectional layout allows the implementation of simple and elegant tracing algorithms.

In the first case, the gc accesses the lock word to get the number of references n (one shift, one mask). If n is the overflow value (big object), then n is retrieved from the virtual table. Finally, the gc simply traces n references in front of the object header.

In the second case, the object instance is reached from its starting point in memory, which might be either a reference field or the object header (if there are no reference fields in this instance). At this point, the gc must find out whether the initial word

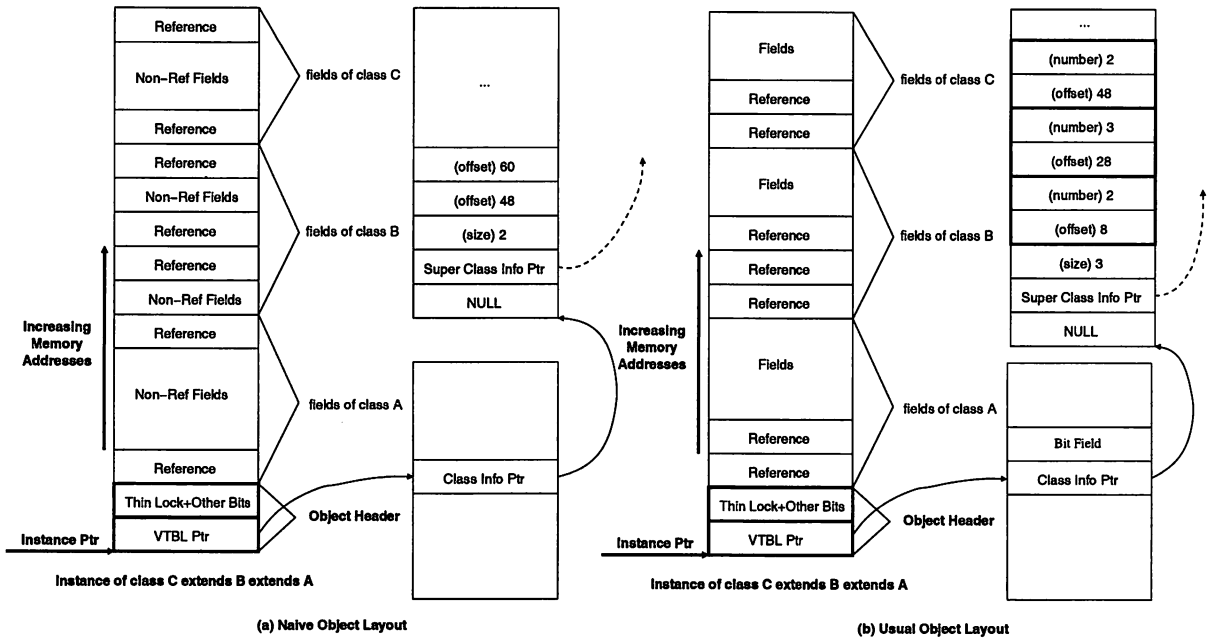


Figure 3: Traditional layout

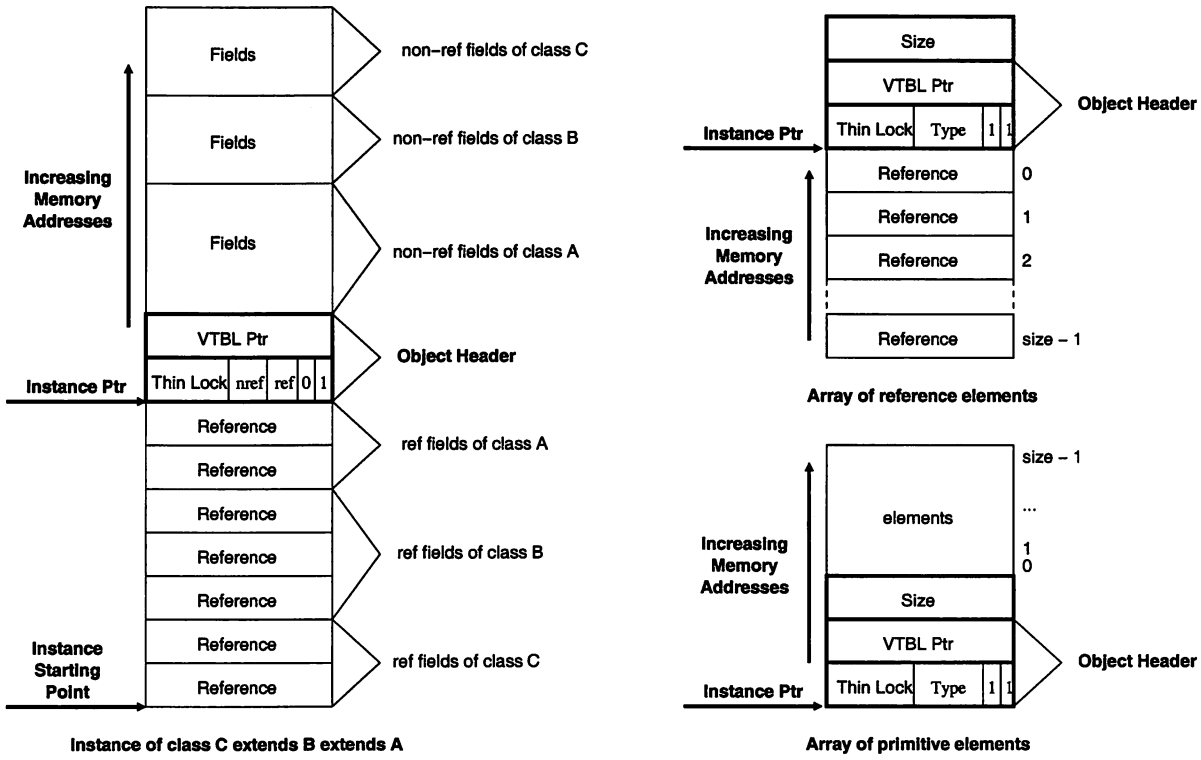


Figure 4: Bidirectional layout

is a reference or a lock word. But, this is easy to find. The gc needs simply check the state of the last bit of the word. If it is one, then the word is a lock word. If it is zero, then the word is a reference.

So, for example, a copying collector, while scanning the *tospace* needs only read words consecutively, checking the last bit. When set to zero, the discovered reference is traced, when set to 1, the number of non-reference field words (encoded in the lock word itself, or in the virtual table on overflow) is used to find the starting point of the next instance.

In summary, using our bidirectional layout, a gc only accesses the following memory locations while tracing: reference fields and lock word, for all instances (objects and arrays), and at most three additional accesses for objects with many fields (virtual table pointer and two words in the virtual table itself).

While our work on bidirectional objects for grouping references is new, we mention some previous related work. The idea of using a bidirectional object layout (without grouping references) has been investigated[25, 28] as a mean to provide efficient access to instance data and dispatch information in languages supporting multiple inheritance (most specifically C++). In [14], Bartlett proposed a garbage collector which required grouping pointers at the head of structures; this was not achieved using bidirectional structs.

6.2 Sparse Interface Virtual Tables

In this subsection, we present a virtual table layout that eliminates the overhead of interface method lookup over normal virtual method lookup.

This enhancement addresses a problem raised by multiple inheritance of interfaces in Java. The virtual machine instruction set contains an *invokeinterface* instruction, used to invoke interface methods. A common technique to implement this instruction is to prepare multiple virtual tables for each class: a main virtual table used for normal virtual method invocation, and one additional virtual table for each interface directly or indirectly implemented by the class[5]. Each method declared in an interface is given an index within its virtual table. After preparation, each *invokeinterface* has two arguments: an interface number, and a method index. On execution, the *invokeinterface* instruction operates its method lookup in two steps. It first lookups up the appropriate virtual table (using linear, binary, or hashed search), then it retrieves the method pointer in a single operation from the virtual table entry located at the given method index. This interface lookup procedure has the following overhead over normal virtual method lookup: it needs to do a search to find the appropriate virtual table. It would be possible to implement a constant time lookup using *compact encoding*[31], but unfortunately, dynamic class loading requires updating this information dynamically, which is difficult to do in a multi-threaded Java environment. Our approach is simple, and does not require dynamic recomputation of tables or code rewrite.

The idea of maintaining multiple virtual tables in case of multiple inheritance is reminiscent of C++ implementations[19]. But, Java's multiple inheritance has a major semantic difference: it only applies to interfaces which may only declare method signatures without providing an implementation. Furthermore, if a Java class implements two distinct interfaces which declare the same method signature, this class satisfies both interfaces by providing a single implementation of this method. C++ allows the inheritance of distinct implementations of the same method signature.

We take advantage of this important difference to rethink the appropriate data structure needed for efficient interface method lookup. Our ideas originate from previous work on efficient method lookup in dynamically typed OO languages using of *selector-indexed dispatch tables*[12, 18, 30]. We assign a globally unique increasing index[7] to each method signature declared in an interface. A method signature declared in multiple interfaces has a single index. When the virtual table of a class is created, we also create an *interface virtual table* that grows down from the normal virtual table. This interface virtual table has a size equal to the highest index of all methods declared in the direct and indirect super interfaces of the class. For every declared super interface method, the entry at its index is filled with the address of its implementation. Interface invocation is then encoded with the *invokeinterface* instruction, and a single interface method index. The execution of *invokeinterface* can then proceed at the *exact same* cost as an *invokevirtual*.

[7] In reality, we use a decreasing index, starting at at -1, to allow direct indexing in the *interface virtual table*.

The interface virtual table is a sparse array of method pointers. As more interfaces are loaded, with many interface method signatures, the amount of free space in interface virtual tables grows. The traditional approach has been to use table compression techniques to reduce the amount of free space. However, these techniques are poorly adapted to concurrent and dynamic class loading environments like the Java virtual machine, as they require dynamic recompilation.

Our approach differs. Instead of compressing interface virtual tables, we simply return the free space in them to the related class loader memory manager (see section 5.4). This memory is then used to store all kinds of other class loader related data structures. In other words, we simply recycle the free space of sparse interface virtual tables within a class loader. The layout of interface virtual tables is illustrated in Figure 5.

As interface usage in most Java programs range from very low to moderate, we could argue that it is unlikely that the free space returned by interface virtual tables will grow faster than the rate at which it is recycled. However, in order to handle pathological cases, we also provide a very simple technique, which incurs no runtime overhead, to limit the maximal growth of interface virtual tables. To limit this growth to N entries, we stop allocating new interface method indices as soon as index N is given. Then, new interface method signatures are encoded using traditional techniques. The trick to make this work is to encode interface calls differently, based on whether the invoked method signature has been assigned an index or not. The traditional technique used to handle overflow can safely ignore all interface methods which have already been assigned an index.

6.3 Improved Thin Locks

Our final enhancement improves upon Onodera's bimodal field locking algorithm[26], a modified version of Bacon's thin lock algorithm[13], but without busy-wait transition from light to heavy mode.

Bacon's thin lock algorithm can be summarized as follows. Each object instance has a one lock word in its header[8]. To acquire the lock of an object, a thread uses the *compare-and-swap* atomic operation to compare the current lock value to zero, and replace it with its thread identifier. If the lock value isn't zero, this means that either the lock is already inflated, in which case a normal locking procedure is applied, or the lock is thin and is already acquired by some thread. In the latter case, if the owning thread is the current one, a nesting count (in the lock word) is increased. If the owning thread is not the current one, then there is contention, and Bacon's version of the algorithm busy-waits, spinning until it acquires the lock. When it is finally acquired, it is inflated. Unlocking non-inflated locks is simple. On each unlock operation, the nesting count is decreased. When it reaches 0, the lock byte is replaced by zero, releasing the lock.

The advantages of this algorithm are that a single atomic operation is needed to acquire a thin lock in absence of contention, and more importantly, no atomic operation is required to unlock an object[9].

Onodera eliminates the busy wait in case of contention on a thin lock, using a single additional bit in each object instance. The role of this *contention bit* is to indicate that some other thread is waiting to acquire the current thin lock. Onodera's algorithm differs from the previous algorithm at two points. First, when a thread fails to acquire a thin lock (because of contention), it acquires a fat monitor for the object, sets the contention bit, checks that the thin lock was not released, then puts itself in a waiting state. Second, when a thin lock is released (e.g. lock word is replaced by zero), the releasing thread checks the contention bit. If it is set, it inflates the lock, and notifies all waiting threads[10].

The overhead of Onodera's algorithm over Bacon's is the contention bit test on unlocking, a fairly simple non-atomic operation, and the one bit per object. This bit has the following restriction: it must not reside in the lock word. This is a problem. It is important to keep the per object space overhead as low as possible, as Java programs tend allocate many small objects. It is now common practice to use 2 word headers in object instances; one word for the virtual pointer, and the second for the lock and other information. The contention bit cannot reside in either of these two words (putting it in the virtual table pointer word would add execution overhead to

[8] Only 24 bits of that word are used for locking on 32 bit systems. 8 bits remain free for other uses.

[9] Unlike Agesen's recent *meta-lock* algorithm[9] which requires an atomic operation for unlocking objects.

[10] This is a simplified description. Please refer to the original paper[26] for details.

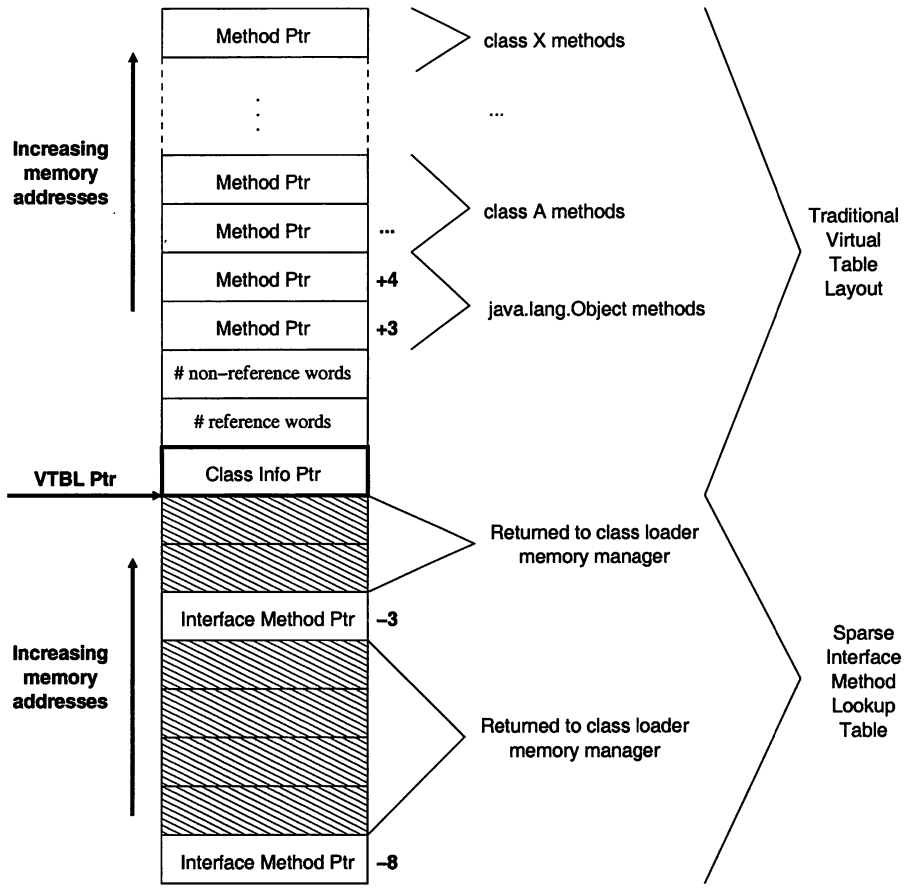


Figure 5: Virtual table layout

method invocation, field access, and any other operation dereferencing this pointer). As objects need to be aligned on a word multiple (for the atomic operation to work), this one bit overhead might well translate into a whole word overhead for small objects. Also, it is likely that the placement of this bit will be highly type dependent, which complicates the unlocking test.

Our solution to this problem is to put the *contention bit* in the thread structure, instead of in the object instance. This simple modification has the advantage of eliminating the per object overhead while maintaining the key properties of the algorithm, namely, fast thin lock acquisition with a single atomic operation, fast thin lock unlocking without atomic operations, and no busy-wait in case of contention.

We modify Onodera's algorithm as follows. In SableVM, each thread has a related data structure containing various information, like stack information and exception status. In this structure, we add the contention bit, a *contention lock*[11], and a linked list of (waiting thread, object) tuples. Then we modify the lock and unlock operation as described in the following two subsections.

6.3.1 Modifications to the lock operation

The lock operation is only modified in the case of contention on a thin lock.

When a thread x_t fails to acquire a thin lock on object z_o due to contention (because thread y_t already owns the thin lock), then (1) thread x_t acquires the contention lock of the owning thread (y_t), and (2) sets the contention bit of thread y_t, then (3) checks that the lock of object z_o is still thin and owned by

[11] The contention lock is a simple non-recursive mutex.

thread y_t. If the check fails, (4a) the contention bit is restored to its initial value, the contention lock is released and the lock operation is repeated. If the check succeeds, (4b) the tuple (x_t, z_o) is added to the linked list of thread y_t, then thread x_t is put in the waiting state (temporarily releasing the contention lock of thread y_t, while it sleeps). Later, when thread x_t wakes up (because it was signaled), it releases the re-acquired contention lock and repeats the lock operation.

6.3.2 Modifications to the unlock operation

The unlock operation is modified to check the contention bit of the currently executing thread. This check is only done when a lock is actually released (as locks are recursive), after releasing the lock.

When the lock of object b_o is released by thread y_t, and if the contention bit of thread y_t is set, then (1) thread y_t acquires its own contention lock, and (2) iterates over all the elements of its tuple linked list. For each tuple (x_t, z_o), if $(z_0 = b_o)$, thread x_t is simply signaled. If $(z_o \neq b_o)$, the lock of object z_o is inflated[12] (if it is thin), then thread x_t is signaled. Finally, (3) thread y_t empties its tuple linked list, clears its contention bit, and releases its contention lock.

7 Experimentation

We are conducting the following experiments, to evaluate the various memory management and performance enhancement strategies.

- Implementation of both standard and bidirectional instance layout, and comparison of the tracing speed of SableVM's copying collector on both layouts.
- Measure the memory overhead of *sparse interface method lookup tables* in application benchmarks. Test, using micro benchmarks, SableVM's behavior in presence of pathological cases.
- Measure the size of *class loader memory fragments* returned to the memory manager for recycling. Measure how much of this memory gets effectively reused. Explore the possibility of not managing these fragments if the storage they require is insignificant.
- Evaluate relative speed of SableVM compared to the speed of other Java virtual machines running on Linux, using a set of standard benchmarks, and some micro benchmarks.

The results of these experiments can be found on the following web site [6].

8 Conclusion and Future Work

In this paper, we have presented SableVM, a framework for testing high-level performance enhancements and extensions to the Java virtual machine. SableVM is written in portable C with minimal system dependecies.

The main goal of the SableVM project was the design and implementation of an open-source virtual machine suitable for research which is easy to modify, can simply handle language and bytecode extensions, and also provides a testbed for various implementation strategies.

Particularly, we have introduced in this paper new high-level techniques usable by any Java virtual machine (including JITs, and hybrid systems) to support efficient execution of Java bytecode.

More specifically, we have introduce a bidirectional object layout that groups reference fields, and we showed how this layout makes tracing objects and arrays simple and efficient.

We also introduced a sparse interface virtual table layout adapted to the dynamic class loading facility of Java, which reduces the cost of an invokeinterface instruction to that of an invokevirtual. We also demonstrated that the sparse representation need not waste memory, because unused holes in the interface table could be recycled and used by the class loader memory manager.

Our last performance enhancement technique was an improvement on thin locks. We introduced a simple algorithm and related data structures that

[12] Notice that thread y_t necessarily owns the lock of object z_o, as a only one lock (on object b_o) has been released by thread y_t since it last cleared its contention bit and emptied its tuple list.

eliminate busy-wait in case of contention on a thin lock. This strategy incurs no space overhead on object instances.

Other groups have expressed an interest in adding other components to the VM, including a JIT compiler. We encourage such collaboration. The SableVM source is publicly-available at: `http://www.sablevm.org/`.

References

[1] Classpath. www.classpath.org/.

[2] GCJ. sources.redhat.com/java/.

[3] Harissa. www.irisa.fr/compose/harissa/harissa.html.

[4] HotSpot. java.sun.com/products/hotspot/whitepaper.html.

[5] Kaffe. www.kaffe.org/.

[6] SableVM. www.sablevm.org/.

[7] Toba. www.cs.arizona.edu/sumatra/toba/.

[8] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a Just-in-Time Java compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 280–290. ACM Press, 1998.

[9] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 207–222. ACM Press, November 1999.

[10] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, October 2000.

[11] Bowen Alpern, Dick Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, and John J. Barton. Implementing Jalapeno in Java. In *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 314–324. ACM Press, November 1999.

[12] Cox B. *Object-Oriented Programming: An evolutionary Approach.* Addison-Wesley, 1987.

[13] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–268. ACM Press, June 1998.

[14] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88.2, Digital – Western Research Laboratory, 1988.

[15] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 49–70. ACM Press, October 1989.

[16] Michał Cierniak, Guei-Yuan Lueh, and James N. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 13–26, Vancouver, British Columbia, June 2000. ACM Press.

[17] Peter Deutsch and Alan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302. ACM Press, January 1984.

[18] Karel Driesen. Selector table indexing & sparse arrays. *SIGPLAN Notices: Proc. 8th Annual Conf. Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, 28(10):259–270, September 1993.

[19] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, December 1990.

[20] Anton M. Ertl. A portable Forth engine. www.complang.tuwien.ac.at/forth/threaded-code.html.

[21] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, 2000.

[22] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management.* Wiley, 1996.

[23] Andreas Krall. Efficient JavaVM Just-in-Time compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 205–212. IEEE Computer Society Press, October 1998.

[24] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, second edition, 1999.

[25] Andrew C. Myers. Bidirectional object layout for separate compilation. In *OOPSLA '95 Conference Proceedings: Object-Oriented Programming*

Systems, Languages, and Applications, pages 124–139. ACM Press, October 1995.

[26] Tamiya Onodera and Kiyokuni Kawachiya. A study of locking objects with bimodal fields. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 223–237. ACM Press, November 1999.

[27] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300. ACM Press, June 1998.

[28] William Pugh and Grant Weddell. Two-directional record layout for multiple inheritance. *ACM SIGPLAN Notices*, 25(6):85–91, June 1990.

[29] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[30] Jan Vitek and R. Nigel Horspool. Taming message passing: Efficient method look-up for dynamically typed languages. In *Object-Oriented Programming, Proceedings of the 8th European Conference ECOOP'94*, volume 821 of *Lecture Notes in Computer Science*, pages 432–449. Springer, July 1994.

[31] Jan Vitek, R. Nigel Horspool, and Andreas Krall. Efficient type inclusion tests. In *OOPSLA '97 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 142–157. ACM Press, October 1997.

[32] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioğlu, and Erik Altman. LaTTe: A Java VM Just-in-Time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 128–138. IEEE Computer Society Press, October 1999.

Dynamic Type Checking in Jalapeño

Bowen Alpern *Anthony Cocchi* *David Grove*
IBM T. J. Watson Research Center
Yorktown Heights, NY, 10598
alpern@watson.ibm.com *tony@watson.ibm.com* *groved@us.ibm.com*
http://www.research.ibm.com/jalapeno

Abstract

A Java virtual machine (JVM) must sometimes check whether a value of one type can be can be treated as a value of another type. The overhead for such dynamic type checking can be a significant factor in the running time of some Java programs. This paper presents a variety of techniques for performing these checks, each tailored to a particular restricted case that commonly arises in Java programs. By exploiting compile-time information to select the most applicable technique to implement each dynamic type check, the run-time overhead of dynamic type checking can be significantly reduced.

This paper suggests maintaining three data structures operationally close to every Java object. The most important of these is a display of identifiers for the superclasses of the object's class. With this array, most dynamic type checks can be performed in four instructions. It also suggests that an equality test of the run-time type of an array and the declared type of the variable that contains it can be an important short-circuit check for object array stores. Together these techniques result in significant performance improvements on some benchmarks.

1 Introduction

A *type check* is the determination of whether a value of one type, hereafter the *right-hand side* or *RHS* type, can legally be assigned to a variable of a second type, hereafter the *left-hand side* or *LHS* type. If so, the RHS type is said to be a *subtype* of (or *consistent* with) the LHS type and the LHS type is said to be a *supertype* of the RHS type. More informally, the types are said to *match*.

Java [9] is a strongly-typed programming language. Almost all type checking is static: done first by a Java source-to-bytecode compiler and then verified by a Java virtual machine (JVM) when classes are loaded. However, Java's object oriented type system makes provision for some run-time type checking. Explicitly casting a value of one type to another (the `checkcast` bytecode), or testing whether such a cast would succeed (`instanceof`), clearly requires such a check. So does storing an object in an array (`aastore`), dispatching a method through an interface (`invokeinterface`), or catching an exception (`athrow`).

While dynamic type checking overhead is unlikely to be the dominant performance characteristic on many Java programs, it is significant enough on some that it is important that care be taken as to how such tests are computed. This paper presents techniques for exploiting information available at compile time to generate efficient code sequences.

The next section presents some background on Jalapeño. Section 3 investigates the cases (`instanceof`, `checkcast`, and `invokeinterface`) where the putative type is known at compile time. Section 4 deals with the cases (`aastore` and `catch` blocks) where this type cannot be determined until run time. Section 5 shows that careful implementation of dynamic type check can result in significant performance improvements on some benchmarks. Section 6 reviews other approaches to fast dynamic type checking. And, section 7 concludes.

2 Jalapeño Background

The work reported here was undertaken in conjunction with the development of the Jalapeño JVM [1] at IBM's T. J. Watson Research Center. Jalapeño is primarily designed for servers.[1] It is written in Java, but (rather than running on top of another JVM) it runs directly on PowerPC-based multiprocessors running the AIX operating system [2].

Jalapeño does not interpret bytecodes. Rather, bytecodes are compiled into machine code and executed directly.[2] Jalapeño has a *baseline* compiler, which produces inefficient machine code very quickly, and an *optimizing* compiler, which can be used to get efficient machine code for selected methods. While the Jalapeño system can be used in a variety of configurations, this paper focuses on a configuration in which all methods are initially baseline compiled, and those observed to be computationally intensive or frequently executed are recompiled by the optimizing compiler. To implement dynamic type checking, the baseline compiler emits code invoking methods that implement the techniques presented in the remainder of the paper, while the optimizing compiler aggressively inlines them.

The next subsection provides a brief overview of the Jalapeño object model. The following subsections describe the prior handling of dynamic type checking by the two compilers. The final subsection deals with the interactions of dynamic class loading and dynamic type checking.

2.1 Jalapeño object model

Types in Java come in three flavors: *primitive* (*e.g.* `int` and `float`), *class* (*e.g.* `Object`, `Truck`, and `Serializable`), and *array* (*e.g.* `int[]`, `Object[][]`, and `Truck[]`). Classes are either *proper classes* (*e.g.* `Object` and `Truck`) or *interfaces* (*e.g.* `Serializable`). Jalapeño represents Java types as objects of the class VM_Type. This class is abstract with three final subclasses: VM_Primitive, VM_Class, and VM_Array. A field of the VM_Class class distinguishes proper classes from interfaces.

Jalapeño also maintains a *type information block* (TIB) for each type. A TIB is an Object array that contains an *interface method table* (IMT) and a *virtual method table* (VMT) for the type. The first slot of the TIB is a reference to its VM_Type object. The work reported in this paper introduces three new slots in the TIB.

Every object has a header. All object headers contain a reference to the TIB for the object's type. (Object headers for arrays contain their length.) Another reference to each TIB is kept in an array of static values called the *Jalapeño table of contents* (JTOC). During execution, a dedicated register holds a pointer to the base of the JTOC. Figure 1 depicts the data structures associated with the TIB of a prototypical class `Truck`. The purpose of the (new) second, third, and fourth slots in the TIB will be discussed in Sections 3.1, 3.2, and 4 respectively. (Note that this picture is a simplification of the Jalapeño object model [1]).

The optimizing compiler's intermediate representation includes special-purpose operators to explictly represent manipulations of the Jalapeño object model. Thus, its entire suite of classical optimizations (common subexpression elimination, loop invariant code motion, etc.) can easily be applied to operations such as loading the TIB from an object (or from the JTOC) or loading the contents of the three new TIB slots.

2.2 Prior baseline dynamic type checking

Jalapeño's baseline compiler emitted code to call Java methods to handle each of the dynamic type-checking bytecodes. These methods in turn called a central `isAssignableWith` method. This method took two VM_Types and returned true, if a value of the second could be assigned to a variable of the first, and false, otherwise.

Occasionally, `isAssignableWith` needed to load classes dynamically. Class loading updates Jalapeño's global data structures, which requires holding a global lock. Acquiring the

[1] The main implications of being a *server* JVM are an obsession with performance, particularly on multiprocessors, and a relative insensitivity to space considerations.

[2] Hereafter, the term "compilation" will refer to translation from bytecode to machine code unless the translation from source code to bytecode is explicitly indicated.

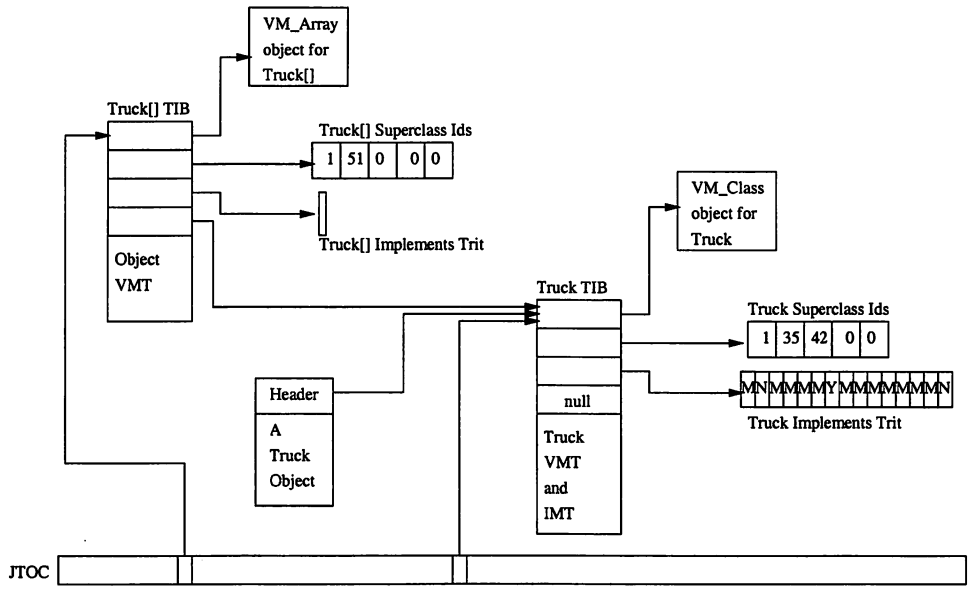


Figure 1: A simplified picture showing the structure of the TIB and associated data structures for the `Truck` class. Truck's TIB is reachable from the JTOC, from the object header of all Truck instances, and from the TIB of Truck[].

global lock to do trivial downcasts and object stores would be costly and would introduce unnecessary contention. Therefore, two short circuit tests are performed to avoid acquiring the lock. The first checks if the two types are equal (and therefore trivially match). If this equality check fails, a two value cache is consulted before acquiring the lock. Every VM_Type object maintains the last RHS type to be found to be assignable with this LHS type and the last RHS type to be found *not* to be assignable with this LHS type.

2.3 Prior optimizations

The optimizing compiler performs type analysis [10, 5] and constant propagation. This enables many `instanceof` and `checkcast` bytecodes to be completely evaluated at compile time. For instance,

- If the RHS value is known to be null, then `checkcast` succeeds and `instanceof` evaluates to `false`.
- If a `checkcast` follows on the true branch of an `instanceof` test for the same LHS type, the checkcast can be eliminated.
- Similarly, an `instanceof` (or `checkcast`) following a `new` of the same LHS type can be eliminated.

More obscure variations of these idioms are also handled. Similarly, the dynamic type check required as part of interface invocation can often be optimized away. Aggressive inlining, performed in tandem with these analyses, uncovers more opportunities to apply these transformations.

If a dynamic type check cannot be so eliminated, the optimizing compiler inlined code to handle one important special case. If the LHS type is known at compile-time to be a final class, there can only be a type match if the RHS type is exactly the same as the LHS type. In which case, the two types TIBs are at the same address. Figure 2 shows assembly code for the sequence used for this test. The LHS TIB is obtained at a fixed offset from the JTOC.[3] The RHS TIB is loaded from the object header. The two TIB pointers are compared. If they are equal, the types match. This test takes two independent loads and a comparison.

In all other cases, the optimizing compiler emitted code to call the same Java methods as the baseline compiler. The techniques de-

[3] A pointer to the LHS TIB cannot easily be stored as a constant in the compiled code in Jalapeño since a copying collector might move the TIB between compilation and execution.

```
L    R1,LHSoffset(JTOC)    // get TIB address of LHS from JTOC
L    R2,TIBoffset(RHS)     // get TIB address from RHS Object
CMP  R1,R2                 // check for equality
```

Figure 2: Assembly code for type equality test.

scribed in the following sections are now inlined by the optimizing compiler to handle these cases.

2.4 Dynamic class loading

If the LHS type of a `checkcast` or `instanceof` bytecode has not loaded when the bytecode is compiled, the compiler cannot determine whether the LHS type is a proper class or an interface. Furthermore, the first execution of the generated code may need to load, resolve, and initialize this type. To handle this situation both compilers emit code that invokes a helper method.

After the first call, the overhead of calling the helper method is superfluous, the type will have been loaded. It is not expected that superfluous calls to the helper method will be frequently executed. Any method containing a frequently executed call site in baseline compiler code will be recompiled by the optimizing compiler. By the time the code is recompiled, any class that has not yet been loaded is unlikely to be referenced often in the future.[4]

Hereafter, it is assumed that the LHS type has been loaded at compile time.

3 When the LHS type is known

The `instanceof`, `checkcast`, and `invokeinterface` bytecodes name the type that the RHS value is supposed to match. This section examines how this information can be exploited if the named class has been fully loaded at compile time.[5] The next subsection addresses the case that the LHS type is a proper class. Subsequent subsections consider interfaces and arrays.

3.1 Proper classes

This section considers by far the most prevalent case of dynamic type checking: the LHS is known to be a proper class that has been loaded at compile time. To handle this case, Jalapeño maintains two (`short`) fields in each VM_Type: a unique type id and a type depth. The depth of Object (and of primitive types) is 0. The depth of arrays is 1. And, the depth of a class (other than Object) is one more than the depth of its superclass.

Following Cohen [6], one of the new slots in a type's TIB is devoted to a display of superclass ids. The `ith` component of this array (of `shorts`) is the id of the superclass of the type at depth `i`. The id of Object is the zeroth component of this array for every proper class. The class's own id is the depth'th component of its display.

To answer the dynamic type checking question in this case, the RHS TIB is loaded from the header of the RHS object, the RHS superclass ids display is loaded from the TIB, the component of this array corresponding to the depth of LHS is loaded, this value is compared to the id of LHS, the match succeeds if, and only if, these quantities are equal. Both the depth and the id of LHS are compile-time constants.

There is flaw in the above scenario: what if the depth of LHS is greater than the depth of RHS? In this case, the types don't match. But, to check for it requires an additional (semi-independent) load and an additional comparison. This is especially annoying since most classes (particularly those that appear on the left hand side of a dynamic type checking question) have very small depth. Assembly code with and without this checking is shown in Figure 3.

To avoid most bounds check, all superclass

[4] To handle the rare case that a previously unexecuted path becomes the hot path of a method, the helper method could report its caller to the adaptive system's controller [3] every time it is called. If a particular method calls a helper method too often, the controller could schedule it for re-recompilation.

[5] The `instanceof` and `invokeinterface` bytecodes fail if the RHS value is null, `checkcast` succeeds. Often the compiler can infer the result of this test at compile time. It is assumed below that the value is known not to be null.

```
Without array bounds checking
  L    R1,TIBoffset(RHS)      // get TIB adddress from RHS Object
  L    R1,SuperclassIDs(R1)   // get superclass ids display
  L    R1,LHSdepth*2(R1)      // get ID from display
  CMPI R1,LHSid               // compare ids

With array bounds checking
  L    R1,TIBoffset(RHS)      // get TIB adddress from RHS Object
  L    R1,SuperclassIDs(R1)   // get superclass ids display
  L    R2,LengthOffset(R1)    // get array length
  CMPI R2,LHSdepth            // is the array large enough?
  BGE  NOMATCH                // if not, the types don't match
  L    R1,LHSdepth*2(R1)      // get ID from display
  CMPI R1,LHSid               // compare ids
```

Figure 3: Assembly code for subclass test.

ids arrays are padded out to some minimum length[6] (currently six).[7] For any test with LHS depth less than this length, the bounds check can be (and is) omitted. Thus, the bulk of dynamic type checks can be performed in four instructions: three (dependent) loads and a comparison. It is so fast that it does not pay to perform a short-circuit test for the case when LHS and RHS are equal (see Figure 2).

3.2 Interfaces

Closely related to the normal case, but much less prevalent, is the case that the LHS type is an interface.[8] This case occurs for the `invokeinterface` bytecode [12], and for the `instanceof` and `checkcast` bytecodes if the LHS type happens to be an interface.

To handle interfaces, a slot in the TIB is devoted to an array indexed by interface id. The value at an entry in this array tells whether the class implements this interface. Since the result of the question cannot in general be determined before it is asked for the first time,[9] the entries in this array have three possible values: YES, NO, and MAYBE. Whenever a MAYBE value is encountered the proper answer is computed and stored into the array. The values of this *implements trits* array is currently stored as a byte.[10]

Assembly code for this case is shown in Figure 4. In the usual (non-MAYBE) case, the answer may be computed with three dependent load instructions. An additional comparison and usually-untaken conditional branch are needed to account for the MAYBE case. As with the superclass ids display, the need for an array bounds check can be eliminated by padding all implement trits array to a minimum size. However, unlike with proper classes, entries in this fast portion are a scarce resource.[11] When a bounds check is needed, it

[6] Alternatively one could allocate a fixed number of short slots in the TIB to hold the first k entries of the superclass ids display. This has the additional benefit of eliminating a load from the common case test sequence. We chose not to do this in Jalapeño because storing non-reference values in a TIB (which is declared as an `Object[]`) would require special-case extensions to Jalapeño's garbage collectors.

[7] 98% of all the proper classes in our benchmark programs (Table 1), the libraries they use, and Jalapeño itself were covered by using a minimum length of 6. A minimum length to 4 would have covered 92% of them, and a minimum length of 2, 54%.

[8] The number of interfaces is much smaller than the number of proper classes. Most proper classes don't implement any interfaces and many of those that nominally do implement interfaces are, in fact, never used as instances of interfaces. For all these reasons, bytecodes that name interfaces are executed far less often than those that name proper classes.

[9] Java interfaces are not leaded with the class that implements them. Furthermore, either a class or an interface may have changed since the source-to-bytecode compiler established that the class implemented the interface.

[10] If space were a more important consideration, each trit could be encoded in two bits. This would require an additional rotate and mask to unpack its value.

[11] In the current implementation, interface entries are assigned in a first-come, first-served fashion (although we ensure that a few commonly used interfaces such as `java.util.Enumeration` receive a fast entry). In the future, online profile data may be used to determine which interfaces should be assigned to the fast portion of the implements trits array.

```
Without array bounds checking
  L    R1,TIBoffset(RHS)        // get TIB adddress from RHS Object
  L    R1,ImplementsTrits(R1)   // get array of trits from TIB
  L    R1,LHSInterfaceId(R2)    // get trit for this interface
  CMPI R1,1                     //  1 => yes   (class implements intrface)
  BLT  NO                       //  0 => no
  BGT  MAYBE                    // >1 => maybe (further checking required)

With array bounds checking
  L    R1,TIBoffset(RHS)        // get TIB adddress from RHS Object
  L    R1,ImplementsTrits(R1)   // get array of trits from TIB
  L    R2,LengthOffset(R1)      // get length of trits array
  CMPI R2,LHSInterfaceId        // can trits array contain this interface?
  BLE  MAYBE                    // trits array too small => maybe
  L    R1,LHSInterfaceId(R2)    // get trit for this interface
  CMPI R1,1                     //  1 => yes   (class implements intrface)
  BLT  NO                       //  0 => no
  BGT  MAYBE                    // >1 => maybe (further checking required)
```

Figure 4: Assembly code for implements interface test.

requires an additional semi-dependent load of the array length, a comparison, and a usually-untaken conditional branch. If the bounds check fails, the implements trits array is extended and filled in with MAYBEs.

The need to handle the MAYBE case allows for a cute space savings. Most classes never end up of the right hand side of a dynamic type checking question with an interface on the left hand side.[12] We want to avoid allocating a big implements trits array for these classes since it will never be used. (Of course, one cannot tell which classes these are ahead of time.) Therefore all TIBs are initialized with their implements trits slots pointing to a single shared trits array filled with MAYBEs. The routine that handles the MAYBE case checks to see if an object's TIB still points to this shared array and if necessary allocates a new non-shared array before changing the MAYBE to a YES or NO.

A further space savings might be realized by exploiting the observation that equivalence classes of classes could share the same implements trits. All members of an equivalence class derive from the same root class and none of them except the root implement any new interfaces.

3.3 Arrays

The final case where LHS type is known is when it is an array. This case has three distinct subcases depending on whether the innermost element type is a primitive type (or a final class), the Object class, or some other non-final class type.

To support type-checking of arrays, each type is assigned a *dimension* (stored in a `short` field of `VM_Type`): primitive types, -1; class types, 0; and arrays, the number of left brackets ("[") in their descriptors. In effect, the dimension of an array is the number of times it can be subscripted.

For an object to be assignable into a variable that is an array whose innermost element type is primitive, or is a *final* proper class, the object's class must be *identical* to that of the variable: it must be an array of the same dimension with the same innermost element type. In this case, the type equality (figure 2) test serves as the dynamic type check.

This type equality test can be performed for the remaining cases as well. If it succeeds, the types match. Otherwise, further tests are required. If the innermost class of the LHS is a class other than Object, then the RHS must

[12] Instances of only 21% of all types in our benchmark programs (Table 1), the libraries they use, and Jalapeño itself actually appeared as the RHS of a dynamic type checking question when an interface was the LHS.

have the same dimension, and its innermost type must be assignable with that of the LHS. If the innermost class of the LHS is Object, any array of the same dimension whose innermost type is not a primitive type is assignable with it. Furthermore, any array with greater dimension is assignable with it, since any array can be assigned to a variable of type Object.

4 When the LHS type is unknown

This section considers two cases where the LHS type of the dynamic type checking question cannot be determined at compile time. These cases arise when an exception is thrown and when an object is stored in an array.

When an exception (or any other `Throwable`) is thrown, Jalapeño walks up the thread's call stack examining stack frames. In each frame, it determines the call site, finds any try blocks that contain it, and checks to see if the exception (the RHS) type matches the type declared for an associated catch clause (the LHS type). This is a constrained instance of the dynamic type checking question. Both types must be proper subclasses of `Throwable`. The depth and id of the LHS type are obtained from its type object (stored in an object associated with the compiled method). There is a match if, and only if, the entry in the RHS's superclass ids display at the LHS's depth equals the LHS's id.[13]

When an object is stored in an array, a check is performed to make sure that the type of the object is compatible with the element type of the array. It may not be immediately obvious why such a dynamic type check is necessary. After all, Java's source to bytecode compiler (and, at class-load time, the JVM's verifier) guarantee that the type of the object is compatible with the element type of the declared array. The problem is that this declared element type may be less restrictive than the runtime element type.

Consider Figure 5. Variable `x` is declared to be an array of `Object` and variable `y` is declared to be a `Petunia`. At compile time (and classload time), the assignment `x[0] = y;` looks fine because `Petunia` is a subclass of `Object`. However, suppose that `makeObjectArray` returns an array of `Truck`. Unless `Petunia` is a subclass of `Truck` (or `y` is null), the assignment is illegal. This can only be determined at runtime.

```
Object [] x    = makeObjectArray();
Petunia   y    = makePetunia();
          x[0] = y;
```

Figure 5: Object array stores require checking.

This is unfortunate. In the first place, in almost any Java program, although object array stores may be fairly frequent, they fail extremely rarely, if at all. Furthermore, in a typical Java program, almost all of such stores are perfectly innocuous: either the type of the object being assigned is the element type of the array being assigned into, or the runtime type of the array is the same as the declared type of the variable that holds it. To make the former case easy to check, a slot in the TIB is devoted to the element type TIB. (This entry is null in TIBs for non-array types.) To make the short-circuit test the TIBs of both the LHS and the RHS are loaded. Then the LHS's element type TIB is loaded. If this is equal to the RHS's TIB, the match succeeds.

The latter case is even easier, *if* the declared type of the variable containing the array is available at compile-time. The TIB for the LHS's declared type is loaded at a fixed offset from the JTOC. If this is equal to the LHS's runtime TIB, the match succeeds. This short-circuit test is especially nice in that it is oblivious to the type of the RHS value. Thus, the test can be hoisted out of a loop in which some or all of the entries of an array are assigned values.

There are two ways the declared type of an array could inadvertently get estranged from the bytecode. If the array is stored in a pure local variable (as opposed to a parameter or a field), Java's source-to-bytecode compiler has thrown away the declared type (that it must be a subtype of `Object[]` can be inferred). More prosaically, an array of a particular class could get upcast to an array of a more general type (usually `Object`) as a side-effect of

[13] This scheme works even if the LHS type has not been loaded. The depth of such a type appears to be zero (because it has not yet been computed), but the type already has a unique id. Since this id is not the id of Object (slot 0 of the RHS's display) the match fails as it should.

being passed to some generic service method. A prime example is the `arraycopy` method of java.lang.System. Sometimes, such situations can be ameliorated by inlining the offending service method. Notice, that even if the information about the declared type of the LHS has been lost, this test degenerates into a short-circuit test for the runtime LHS type being `Object[]`, an important subtest in its own right.

In the cases that these two short-circuit tests fail, the most prevalent situation is that the LHS array is an array of some proper class (rather than an array of an interface or an array of arrays). A type match can be detected fairly quickly. The LHS element type is loaded from its TIB (available from the failed short-circuit tests). The depth and id fields can be loaded from this type. If the depth'th entry in the RHS's superclass ids display is the element type id, the types match. Otherwise, either the types don't match, or the LHS is an array of interfaces or an array of arrays. These final cases so rare that they can be safely be handled by a helper method without measurable impact on performance.

5 Performance

Jalapeño can be deployed in a multitude of variations. Key discriminates include: the garbage collector (copying or not, generational or not), the compiler (baseline or optimizing, level of optimization) to be used on methods of classes included in Jalapeño's boot image, the compiler to be used on methods of classes loaded dynamically, whether such methods should be compiled immediately when their class is loaded, and whether, under what circumstances, and how methods should be recompiled.

For our experiments, we used a configuration that currently obtains the best Jalapeño performance on the SPECjvm98 benchmarks. It uses a copying, non-generational garbage collector. The optimizing compiler (at optimization level 2) is used to statically compile all methods in the boot image. The first time a dynamically loaded method is invoked, it is compiled using the baseline compiler. Methods observed via online profiling to be computationally intensive or frequently called are selected for recompilation with the optimizing compiler by Jalapeño's adaptive optimization system [3].

The performance impact of the changes to dynamic type checking were evaluated using the SPECjvm98 [7] benchmarks and the Jalapeño optimizing compiler [4]. (Table 1 provides a description of each benchmark, the number of classes that comprise the benchmark, and the size, in bytes, of its class files.) The best elapsed time from 10 runs *during a single JVM execution* of each benchmark is reported. The size 100 (large) inputs were used for the SPECjvm98 benchmarks. The time for the optimizing compiler to compile itself (roughly 80,000 lines of Java source code) is shown as the `opt-compiler` benchmark. Results were obtained on an IBM 43P Model 140 with one 333MHz PPC604e processor and 512MB of main memory running AIX v4.3.

Not surprisingly, the new dynamic type checking implementation only improves the execution times of those benchmarks that perform significant amounts of dynamic type checking: `opt-compiler`, `javac`, and `jess`. Interestingly, these are also the three largest (and arguably the most object-oriented) programs in our benchmark suite. Overall, the largest improvement comes from efficient dynamic type checking for proper classes, although array store checks and interfaces are also factors on some benchmarks.

6 Related work

We are unaware of other work that attempts to exploit the particularities of Java's type system to expedite dynamic type checking. Krall *et.al.* [11] review, and supercede, earlier work on dynamic type checking in a general multiple-inheritance environment.

Oberon [14] is an object oriented language with a simple inheritance model based on type extensions [13]. Cohen [6] presents a display-based technique for constant-time dynamic type checking in this setting. This work did not pad displays to avoid the array bounds checks. Pfister, et. al. [8] pad displays (to eight elements) but restricts the maximum depth of inheritance. Jalapeño does not have such a maximum, rather it uses a minimum display size (6) to eliminate the bounds check in almost all cases.

Benchmarks	Description	Number of Classes	Class File Size (in bytes)
`compress`	Lempel-Ziv compression algorithm	12	17,821
`jess`	Java expert shell system	150	396,536
`db`	Simple memory resident database	3	10,156
`javac`	JDK 1.0.2 Java compiler	175	561,463
`mpegaudio`	Decompression of audio files	54	120,182
`mtrt`	Two-thread ray-tracing algorithm	25	57,859
`jack`	Java parser generator	55	130,889
`opt-compiler`	Jalapeño optimizing compiler	393	1,378,292

Table 1: The benchmark suite. The first seven rows are the SPECjvm98 benchmarks.

Benchmarks	Prior DTC	New DTC	Only Classes	Only Interfaces	Only Arrays	Only AAStore
`compress`	41.16	40.95 (1.01)	41.17 (1.00)	41.82 (0.98)	41.60 (0.99)	42.51 (0.97)
`jess`	24.66	23.09 (1.07)	24.63 (1.00)	23.35 (1.06)	24.53 (1.01)	24.58 (1.00)
`db`	66.66	63.66 (1.05)	67.13 (1.00)	66.70 (1.00)	67.97 (0.98)	63.79 (1.04)
`javac`	42.63	35.33 (1.21)	38.80 (1.10)	41.81 (1.02)	42.91 (1.00)	42.17 (1.01)
`mpegaudio`	22.55	22.24 (1.01)	22.00 (1.03)	23.33 (0.97)	23.89 (0.94)	22.57 (1.00)
`mtrt`	19.42	19.12 (1.02)	19.66 (0.99)	19.04 (1.02)	18.81 (1.03)	19.20 (1.01)
`jack`	35.82	35.43 (1.01)	36.66 (0.98)	35.66 (1.00)	35.88 (1.00)	37.12 (0.96)
`opt-compiler`	146.29	95.76 (1.53)	119.47 (1.22)	153.14 (0.96)	148.37 (0.99)	124.42 (1.18)

Table 2: **Execution times in seconds.** The second column gives the execution times using Jalapeño's prior implementation of dynamic type checking. The third column shows the total impact of the new dynamic type checking implementation. The last four columns show the performance obtained by selectively enabling the new dynamic type checking implementations only for proper classes, interfaces, arrays, and array store checks. The number in parenthesis is the speed relative to the `Prior DTC` configuration.

If a program's type hierarchy were available when the JVM started executing, type checking information could be encoded as a two-dimensional *typecheck* array, the i,j-th entry being on if, and only if, the i-th type were a subtype of the j-th. It would be convenient to keep a row of this array in a slot in i's TIB. Where the LHS type of a type check is known at compile time, the tests would require three dependent loads, a rotate and mask (to unpack the j-th bit), and a comparison. Object array store checks would require one or two extra loads to obtain j's index from its VM_Type object. A major drawback to this approach is that the size of the type check array is N^2 in the number of types.

The typecheck array is mostly sparse. A significant compression arises as a by product of a different type check procedure. Krall *et.al.* [11] propose associating with each type a small set of integers such that RHS is a subtype of LHS if, and only if, the set associated with LHS is a subset of the set associated with RHS. They consider various methods of constructing such sets. The two-dimensional array of answers to these subset questions encodes the type-check array at an impressive space savings.

An obvious drawback to both these schemes is that a Java program's type hierarchy is *not* available *a priori*. Thus, a three-valued array element (as with the implements trits array used above for interfaces) seems to be required. And, as even the number of types is not known in advanced, an array bounds check may also be needed with every type check. Krall *et.al.* give an incremental algorithm for computing their type sets, but only if the sets for a type's immediate parents are known when the type is loaded. Because the interfaces a class implements are not loaded

with it, this condition does not hold for Java.

7 Conclusions

Dynamic type checking can contribute a significant component to the overall runtime of a Java program. It is, therefore, important that the most common instances of such tests be executed as efficiently as possible when they are executed frequently. This paper presents techniques for exploiting the peculiarities of the Java language, and its type system in particular, to perform these tests efficiently.

It suggests maintaining three data structures within easy striking distance of an object.

The first, and most important, of these is a display of the ids of the superclasses of a type. In most cases where the prospective supertype is known at compile time, this allows a dynamic type check to be performed in four instructions. (Padding this array out to a known minimum size eliminates most array bounds checks.) This array is also useful when the prospective supertype is not known at compile time (*e.g.* the `aastore` bytecode).

The second is an array that tells whether the object's class implements a particular interface. Since this question cannot be definitively answered before the first time it is asked, entries in the array must be three valued. Dynamic type checking questions where the prospective supertype is an interface type are answered efficiently using this array.

The third data structure is only used to determine whether it is legitimate to store a particular object in a particular array. It allows quick access to information about the element type of an array from the array object. Jalapeño uses the TIB of the element type, but the class object of the element type could be used instead.

The paper also suggests that testing for the equality of a runtime array with the declared type of the variable that contains it (where possible) is an important short-circuit test for object array stores.

The resultant dynamic type checking system is much more space efficient than those based on two-dimensional typecheck arrays, is more friendly to dynamic class loading (and more space efficient) than those based on compressed typecheck arrays, and, on some benchmarks, performs significantly better than the earlier Jalapeño system.

Acknowledgments

Without the ongoing support of the entire Jalapeño group, this work would not have been possible.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[2] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Derek Lieber, Stephen Smith, and Ton Ngo. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, 1999.

[3] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 2000.

[4] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.

[5] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizating dynamically-typed object-oriented programs. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 150–164, 1990.

[6] Norman H. Cohen. Type-extension type tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems*, 13(4):626–629, October 1991.

[7] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1998.

[8] Beat Heeb Cuno Pfister and Josef Templ. Oberon technical notes. Research Report 156, Eidgenossische Techniscle Hochschule Zurich- Departement Informatik, March 1991.

[9] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[10] Ralph Johnson. TS: An optimizing compiler for smalltalk. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 18–26, 1988.

[11] Andreas Krall, Jan Vitek, and R. Nigel Horspool. Near optimal hierarchical encoding of types. In *Proceedings ECOOP '97*, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[12] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification Second Edition.* The Java Series. Addison-Wesley, 1999.

[13] Niklaus Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, April 1988.

[14] Niklaus Wirth and Jurg Gutknecht. *ProjectOberon: The Design of an Operating System and Compiler.* Addison-Wesley, 1992.

Proof Linking: Distributed Verification of Java Classfiles in the Presence of Multiple Classloaders

Philip W. L. Fong
Simon Fraser University, B.C., Canada
pwfong@cs.sfu.ca
Robert D. Cameron
Simon Fraser University, B.C., Canada
cameron@cs.sfu.ca

Abstract

To offload the computational burden of bytecode verification within Java Virtual Machines (JVM), distributed verification systems may be created using any one of a number of verification protocols, based on such techniques as proof-carrying code and signed verification by trusted authorities. This paper advocates the adoption of a previously-proposed mobile code verification architecture, proof linking, as a standard infrastructure for performing distributed verification in the JVM. Proof linking not only supports both CLDC-style and signature-based distributed verification protocols, but it also provides interoperability between the two. To ground our work in the real-world requirements of Java bytecode verification, we also extend previous work on proof linking to handle multiple classloaders.

1 Introduction

Security is the cornerstone of trustworthy mobile code systems such as that of Java. In accepting arbitrary mobile code from unknown and potentially untrustworthy sources, a Java Virtual Machine (JVM) enforces type safety—the first line of defence in mobile code security—through a a link-time bytecode verification process. The bytecode verifier performs dataflow analysis and various structural analyses to guarantee that untrusted classfiles can be linked into the JVM without producing type confusion. We call this protection mechanism, in which a static code verification procedure is invoked dynamically by the runtime environment, *proof-on-demand.*

Proof-on-demand is conceptually simple and allows the JVM to take full responsibility for assuring type safety even in the presence of dynamically generated code. However, proof-on-demand imposes a considerable computational burden on the JVM. The *link-time overhead* is significant enough that some authors hyperbolically compare it to a denial-of-service attack [12, p. 110]. Moreover, the *architectural complexity* of the JVM is greatly increased by the coupling of complex verification logic with lazy dynamic linking [7]. Compounding these concerns, the bytecode verifier also adds significantly to the JVM's *memory footprint* [16, Sec. 5.3.1].

Future computational platforms will likely include a vast array of small information appliances that have limited computational resources and demanding response-time requirements. Downloaded mobile code will continue to be popular to provide short-lived system extensions (see, for example, the mobile code language WMLScript [17] for the Wireless Application Protocol). With its stability and widespread acceptance, the Java platform—and specifically realizations thereof based on the Connected, Limited Device Configuration (CLDC) specification [16]—will likely become a major infrastructure for hosting mobile programs in small devices. In these contexts, however, the high resource requirements and architectural complexity of proof-on-demand implementations may become intolerable. The CDLC specification has hence rejected the proof-on-demand approach.

Future systems will also likely see additional forms of run-time verification to provide enhanced levels of protection. As the pervasiveness of mobile code hosting environments increases, so too do the vulnerabilities and the potential consequences of these vulnerabilities. To counteract this, attention will turn to safety properties that go beyond simple "type safety" in ensuring system integrity. The complex program analyzers necessary to verify these additional properties may well become impractical even for a standard JVM, let alone a CLDC-compliant device.

To address these issues, some or all of the verification burden may be offloaded to parties other than the mobile code hosting environment. This gives rise to a *distributed verification system*, in which a mobile code runtime environment shares some or all of its verification burden with certain remotely located facilities. Each facility interacts with the hosting environment by means of a *verification protocol*. A distributed verification system may in fact employ distinct verification protocols for different code units provided that an overall framework for protocol interoperation is defined. An individual verification protocol is thus a fixed scheme that orchestrates the communication and division of labour among the parties involved in the distributed verification of a code unit. For example, proof-on-demand is a trivial verification protocol that assigns the entire verification burden to the host environment.

This paper is a contribution to the ongoing development of a standard distributed verification architecture for mobile code systems in general, and for the JVM in particular. The focus of this paper is twofold:

1. We refine our previously-proposed *proof linking* architecture [5, 7] as an architectural framework for supporting distributed verification of Java bytecode. In particular, our framework provides for highly efficient signature-based verification protocols, as well as interoperability between different verification protocols.

2. We extend our previous work to account for a unique feature of the JVM, namely, multiple classloaders. We articulate how this extension preserves the modularity of our architecture as well as the correctness conditions of the proof linking strategy. We further describe how this extension may be implemented, not only in a CLDC-compliant environment, but also in a standard JVM.

2 Distributed Verification: Related work and major issues

In a distributed verification system, the machine hosting the mobile code runtime environment is called the *code consumer*. The party responsible for construction and distribution of mobile programs is the *code producer*. Code producers and consumers interact in various ways to define a verification protocol.

As an alternative to proof-on-demand, two families of verification protocol have been proposed in the related literature.

Self-Certifying Code. The first protocol family involves augmenting the untrusted code to make it self-certifying. This approach is exemplified in the work on *proof-carrying code* [14, 13]. The protocol proceeds as follows. (i) The code consumers, or possibly an authority representing them, publish a safety policy in the form of a verification-condition generator. Given any mobile program, the generator computes a verification condition that must be shown to be true if the code is to be accepted as safe by consumers. (ii) To distribute a program, a code producer computes the verification condition from the code, proves the condition, and then attaches the proof to the program code when it is distributed. (iii) Upon receiving a mobile program, a consumer recomputes the verification condition, and then checks if the attached proof indeed establishes the verification condition. Execution is granted if proof checking succeeds. Since proof checking is often substantially easier than proof generation, this protocol induces less link-time overhead than proof-on-demand. Furthermore, since proof generation may now be performed once and for all on the producer side, difficult-to-prove safety properties may become affordable.

In application to Java, the essential idea behind proof-carrying code is that the code producer can annotate a mobile program with static analysis results, so that a consumer may use the annotations to avoid performing a full bytecode verification. This idea has been applied to the verification of Java

bytecode in various forms [15, 9, 1], and has further been incorporated in the stack map method of the CLDC specification [16, Sec. 5.3].

Signature-based Methods. A second family of distributed verification protocols is based on a very efficient and well-understood mechanism, namely, signature checking. Execution is granted to code that is signed by a trusted party.

A major objection to these protocols is that, unlike a proof (or other kind of annotation), the semantics of a signature may not be well defined. Thus, there may be no protection against the possibility that signing authorities miscertify. Moreover, celebrity is required in the certification of mobile programs, making it hard for non-established developers to inspire trust.

These objections are nicely addressed by a protocol which we call *proof delegation* [2, 3]. The protocol proceeds as follow. (i) The code consumers, or more likely an authority representing them, publish a safety policy in the form of a static program analyzer that checks if a given mobile program is safe. The analyzer is encapsulated in a trusted coprocessor, for example, having the form factor of a PCMCIA card or a PCI card [8]. Attempts to physically tamper with the encapsulated analyzer or to extract the private encryption key in the hardware will render the hardware dysfunctional, or perhaps clear its memory [4]. The hardware is then distributed to code producers. (ii) To distribute software, a code producer submits mobile programs to the trusted program analyzer, which verifies the safety of the code, and digitally signs it. (iii) Upon arrival at a consumer site, the signature atttached to the program code will be authenticated. The bytecode verification of the proof-on-demand protocol is replaced by a simple and efficient signature-checking primitive.

Using trusted coprocessors, proof delegation physically binds the signature to the formal properties enforced by a static program analyzer, thereby giving a well-defined semantics to the signature. However, in order to support signature-based verification protocols such as proof delegation, two further issues must be addressed.

1. **Conditional Certification.** When a Java classfile is verified remotely, it is only checked against the classes on the producer side. However, Java type safety is a run-time notion, and a classfile is safe only if we check it against the loaded classes on the consumer side. For example, during verification of a classfile, the dataflow analyzer might need to show that class A is a subclass of class B. This fact can only be shown by examining the classes that are already loaded into the consumer's JVM. As a result, a conditional certification semantics for signature is needed. That is, signatures certify that a classfile is safe if specified external dependencies are further validated on the consumer side at link time.

2. **Protocol Interoperability.** A Java developer may use some off-the-shelf components, and write "glue" code to orchestrate their interaction. A common scenario may be that the prefabricated components are already certified using efficient signature-based protocols, while the home-grown connection code is certified by CLDC-style stack maps. A JVM hosting this program will not only need to be fluent in both protocols, but also need to *combine two different kinds of certificate (signatures and stack maps) when accessing the safety of the whole program.* What is needed, then, is a mechanism to hide the details of a module's certificate, and examine only its *certification interface*, which offers us a safe mechanism for combining certificates.

In support of both self-certifying protocols and signature-based methods, this paper proposes a *proof linking* architecture with the following features.

1. Each module is *verified separately*, using its own verification protocol.

2. Certificates are annotated by a well-defined verification interface, namely, a set of *proof obligations and commitments*, which delineate the external dependencies of the certified unit.

3. A lightweight, incremental verification process, namely, *proof linking*, is integrated into JVM's linking algorithm, so that proof obligations can be discharged by commitments.

This architecture closely follows that previously developed to deal with modularity concerns in Sun's

JVM [5, 7]. Here, however, we refine the architecture to addess the needs for conditional certification and protocol interoperability in distributed verification systems.

3 The Proof Linking Architecture

3.1 Assumption, Notations and Modelling

The standard Java classloading semantics uses multiple classloaders to implement namespace partitioning. A loaded class is identified by both its classname and its defining classloader [10]. For the purpose of introducing the proof linking architecture and its application to distributed verification, however, this section makes the simplifying assumption that only a single classloader exists. This assumption will be relaxed in subsequent sections.

As a result of the single-classloader simplification, symbolic class references and loaded classes are each denoted simply by classname. A member named M of a class X with type signature S is identified by the symbolic reference $X::M(S)$. We do not differentiate a symbolic member reference and an actual member of a loaded class.

We model the linking activities of the simplified JVM by a set of *linking primitives*, including, for example, "**load** X", "**verify** X", "**resolve** Y **in** X", "**resolve** $Y::M(S)$ **in** X", and so on. The JVM executes linking primitives in a concurrent, nondeterministic fashion, in accordance with a well-defined partial ordering enforced by the implementation. We call this partial order the *linking strategy* of the runtime environment. Further details are presented in Section 4.

3.2 A Motivating Example

Signature-based verification protocols involve the remote verification of classfiles. Safety is in general a whole-program notion, and thus cannot be established by merely examining a single module. External dependencies (e.g., checking for assignment compatibility, computing the meet of two flow values, and so on) validated remotely may no longer hold when a classfile is linked into the runtime environment. The following example is illustrative.

Suppose class A defines a method $M(S)$. Suppose further that A has a direct subclass B, which in turn has a direct subclass C. Assume that C overrides the method $M(S)$. Say the method $C::M(S)$ contains an `invokespecial` bytecode instruction that delegates the call to method $A::M(S)$. A bytecode verifier is supposed to check that class C is a subclass of class A. If the verification of class C is performed at link time, the bytecode verifier will recursively invoke the classloader to bring in the classfiles for both B and C to validate the required subclassing relationship. However, if the verification of class C is carried out remotely, this strategy will fail for two reasons. First, the remote verifier might not have access to the classfiles for A and B. Second, even if it does, the versions of A and B considered by the remote verifier may not be compatible with the ones actually loaded into the JVM.

This configuration management issue motivates a two-stage verification architecture based on conditional certification and proof linking. In the first stage, remote verification is performed in a modular fashion to generate conditional certificates. These certificates specify the assumptions (dependencies) that must be checked to ascertain the safety of the code unit in question. In the second stage, the JVM performs proof linking to check the asserted conditions in the context of the classes actually loaded.

3.3 First Stage: Modular Verification

Untrusted classfiles are certified by a trusted verifier at a remote site. Instead of endorsing external dependencies that might be invalidated at runtime, the remote verifier computes, for each classfile, a conservative *safety precondition* summarizing the external dependencies that must hold at runtime in order for the classfile to be safe. The safety precondition is here *represented* as a conjunctive set of database queries. In the running example, during the remote verification of classfile C, a trusted verifier will generate the query `subclass(`C`,`A`)` as the `invokespecial` instruction is scanned. The remote verifier may end up generating many such queries. The conjunctive set of all the queries formulated by the remote verification session becomes the safety precondition for endorsing the classfile being considered.

In addition, the remote verifier also schedules each of the queries for evaluation. Each query describes a safety precondition for a particular linking primitive. For example, the query above, `subclass(`C`,`A`)`, is associated with the linking primitive "**resolve** $A::M(S)$ **in** C." In essence, this schedules the subclassing check for evaluation immediately prior to resolution of $A::M(S)$ within class C. Such a query is said to be the *proof obligation* for the associated primitive, representing a condition that must be met if the runtime system is to safely execute the corresponding linking primitive. We also say that a proof obligation is *attached* to its associated primitive.

In order for the runtime system to discharge proof obligations, the remote verifier also computes, for each code unit, a set of clauses called *commitments*. The commitments are ground facts that describe the interface properties of the code unit. For example, during the remote verification of the classfile C, the fact `extends(`C`,`B`)` is generated as one of the commitments. As we shall see below, this fact will be used to satisfy the subclassing query in our example.

Prior to shipping a classfile, the remote verifier will package together the classfile itself, the obligations annotated with the associating primitives, and the commitments. The entire package is signed by the verifier using its private key. The signed package is then distributed to consumers. The obligations, commitments, and the signature can all be packaged inside a classfile using classfile attributes (see the JVM specification for details [11]).

3.4 Second Stage: Proof Linking

When a remotely certified classfile X arrives at the code consumer's site, it is loaded into the JVM. The local "**verify** X" primitive, instead of performing bytecode verification on X, will unpackage the classfile, authenticate the signature, and then process the proof obligations and commitments following the procedure outlined below.

We assume that the runtime environment is equipped with two data structures: (1) an *obligation table* mapping linking primitives to their attached proof obligations, and (2) a *commitment database* holding commitments already known by the JVM. After unpackaging an untrusted classfile, the local "**verify** X" primitive records the newly obtained proof obligations in the corresponding entry of the obligation table, and asserts the commitments into the commitment database.

Subsequently, when a linking primitive is to be executed, the JVM will (1) look up the obligations that are already attached to the primitive, (2) attempt to satisfy each of the obligations by consulting the commitment database (raising an appropriate linking exception if the attempt fails), and (3) perform the action prescribed by the primitive.

To make proof linking more expressive, arbitrary logic programs can be provided as an *initial theory* in the commitment database. For example, the following program can be present in the database to define subclassing as the transitive closure of the `extend/2` relation.

```
subclass(X,X).
subclass(X,Y) :-
  extends(X,Z),
  subclass(Z,Y).
```

After the local verifier has processed the classfiles for B and C, the commitment database may contain the following commitments:

```
extends(C,B).
extends(B,A).
```

When the primitive **resolve** $A::M(S)$ **in** C is to be executed, the JVM will look up its attached obligations, among which the query `subclass(`C`,`A`)` will be found. The JVM then attempts to satisfy this query by consulting the facts and rules within the commitment database. The query succeeds and the primitive is executed (assuming that any other obligations are satisfied as well).

3.5 What We Have Gained

The scheme presented above nicely addresses the need for conditional certification. Even though the remote verifier is unable to validate the external dependencies of a class, it nevertheless can express them as proof obligations. The proof linking mechanism is invoked to discharge the obligations at runtime, when the necessary information has become available.

Proof linking also provides interoperability between distributed verification protocols such as proof delegation, proof-carrying code, and proof-on-demand. Because linking primitives communicate solely by attaching obligations and asserting commitments, they are highly modular. A local verifier may process classfiles annotated with CLDC-style stack maps to generate the appropriate proof obligations and commitments just as a remote verifier would do through bytecode analysis. In the event of a classfile that is neither signed by a remote verifier nor annotated with stack map, the local verifier may itself perform a complete bytecode verification to generate the necessary proof obligations and commitments. The proof linking mechanism checks and discharges obligations from each of these sources in the same way, without any need to know the verification protocol used for a particular classfile. As a result, a Java application could consist of a number of classfiles that are signed remotely, others that are "pre-verified", and still others that are completely uncertified. As long as each verification protocol uses the same *verification interface* for asserted proof obligations and commitments, interoperability is assured.

3.6 Correctness Considerations

That the incremental proof linking procedure is as safe as proof-on-demand is not obvious. Could an obligation be generated after its attachment target is already executed? Could it be possible that checked obligations are falsified by subsequently generated commitments? Is it possible that an obligation fails only because the commitments necessary for satisfying it are generated too late? To address these concenrs we formulate the following three correctness conditions.

1. **Safety.** Every primitive that could attach obligations to another primitive p must be completed before the execution of p begins.
2. **Monotonicity.** Obligations may not be contradicted by subsequently asserted commitments.
3. **Completion.** Let o be an obligation that could be attached to a primitive p. All primitives that could generate commitments contributing to the satisfaction of o must be completed prior to the execution of p.

By reasoning about the partial ordering of primitives, and analyzing the structure of the commitments, obligations and the initial theory, we have formally established the three correctness conditions above for our Java proof linking formulation, and have check the proof with the aid of a mechanical theorem prover [7].

3.7 Implementation

Throughout the presentation above, we have used a simple logic programming notation to represent the proof obligations, commitments, and initial theory. However, this abstract model is presented as such only to clarify ideas and to facilitate the articulation of correctness (see Section 5). By no means are we suggesting that an actual implementation employ a logic programming system in proof linking. Rather, commitments can be optimized as flags and pointers stored in the Class and Method objects inside the JVM. A recursive query such as subclassing could well be realized by a standard tree traversal algorithm. The obligation table need not even exist physically; obligations attached to a **resolve** primitive could be stored in the corresponding constant pool entry. Such optimizations were used in the prototype implementations reported previously [5, 7].

4 Java Proof Linking for Multiple Classloaders

The discussion above assumes a simplified JVM with only one classloader. We relax the assumption in this section, and analyze the interaction between proof linking and dynamic linking in the setting of multiple classloaders. It turns out that a systematic, straightforward set of extensions to the previously proposed model is sufficient to make proof linking work with multiple classloaders. This demonstrates that the proof linking technique is applicable to realistic mobile code environments and is orthogonal to Java's delegation-style classloading.

4.1 Enter Multiple Classloaders

When a Java application attempts to load a class C with a given name X with a classloader L_i, *the ini-*

load $\langle X, J\rangle$	Define a class with name X and defining classloader J.
verify $\langle X, J\rangle$	Using the appropriate verification protocol, assess the safety of the bytecode in loaded class $\langle X, J\rangle$.
bind X^L **to** $\langle X, J\rangle$	Bind the class symbol X in the namespace of classloader L to the loaded class $\langle X, J\rangle$. That is, classloader L becomes an initiating classloader of X.
endorse $\langle X, J\rangle$	Endorse the loaded class $\langle X, J\rangle$ for resolution.
endorse $\langle X::M(S), J\rangle$	Endorse the loaded member $\langle X::M(S), J\rangle$ for resolution.
resolve Y **in** $\langle X, J\rangle$	Resolve the class symbol Y in loaded class $\langle X, J\rangle$.
resolve $Y::M(S)$ **in** $\langle X, J\rangle$	Resolve the member symbol $Y::M(S)$ in loaded class $\langle X, J\rangle$.

Figure 1: The Extended Set of Linking Primitives for Java

tiating classloader of C, L_i may delegate the classloading task to another classloader, which, in turn, might delegate the task to yet another classloader. The classloader L_d that eventually loads and defines C is said to be its *defining classloader*. C is uniquely identified by the pair $\langle X, L_d\rangle$. We also write $X^{L_i} \mapsto \langle X, L_d\rangle$ to indicate the fact that L_i initiates the loading of $\langle X, L_d\rangle$. When a symbolic reference Y is resolved in a class $\langle X, L\rangle$, the classloader L will be used as the initiating classloader for class Y. We identify the member $M(S)$ of a loaded class $\langle X, L\rangle$ by the notation $\langle X::M(S), L\rangle$. Details of Java's classloading mechanism are described elsewhere [11, Chapter 5][10].

4.2 Overview of the Solution Approach

Since a loaded class is identified not only by its classname, but also by its defining classloader, a remote verifier has no way of naming the classes in the commitments and obligations it generates. To deal with this, we extend our scheme by the following set of reformulations:

1. The remote verifier expresses commitments and obligations not in terms of loaded classes, but in terms of symbolic class references.
2. The local **verify** primitives tag the symbolic class references by their initiating classloaders.
3. Name binding events are explicitly modeled as linking primitives generating binding commitments.
4. Translation rules are introduced into the initial theory for explicit resolution of tagged symbolic references into loaded classes using binding commitments.

4.3 Linking Primitives

We begin the discussion of our extended proof linking model by looking at its linking primitives. We define a linking primitive as a self-contained action which never activates other primitives as a subroutine, nor recursively invokes itself. The JVM executes a linking primitive p using the following protocol:

1. Look up the proof obligations that have been attached to primitive p.
2. Attempt to satisfy each of the obligations using the commitments that have been asserted into the commitment database. Raise an appropriate exception if any obligation is unsatisfiable.
3. Perform the action prescribed by the linking primitive.
4. Collect the new proof obligations generated by the primitive, and record the obligations in the corresponding entry of the obligation table.
5. Collect the new commitments generated by the primitive, and assert them into the commitment database.

Our multiple-classloader linking model contains the extended set of linking primitives in Figure 1. We inherit the two **endorse** primitives from our original work. They are auxiliary targets of obligation attachment. Since they do not contribute much to our present discussion, readers may safely identify them with class preparation. Only the **verify** primitives generate obligations, and only the **verify** and **bind** primitives generate commitments.

Two changes to the original primitive set have been made [7, Sec. 4.1]:

1. The **load**, **verify**, **endorse** and **resolve** primitives have been adapted to refer to loaded classes rather than simple class names.

2. A new family of **bind** primitives has been introduced. These model the explicit binding of loaded classes to symbols defined in the local namespace of a classloader. When the JVM binds the loaded class $\langle X, J\rangle$ to the symbol X in an initiating classloader L, the primitive "**bind** X^L **to** $\langle X, J\rangle$" is executed. It is assumed that the JVM will execute at most one "**bind** X^L **to** $\langle X, J\rangle$" for each symbol X in classloader L.

The JVM orders the nondeterministic, concurrent execution of linking primitives according to the constraints prescribed by the linking strategy found in Section 4.7.

4.4 Static Type Rules

The static type rules for Java under the single classloader assumption have been presented previously [7, Figure 6]. A straightforward mechanical translation to replace classnames with loaded class notations adapts these rules for the multiple classloader case. For example, consider the subclassing rule mentioned above.

```
subclass(X, X).
subclass(X, Y) :-
  extends(X, Z),
  subclass(Z, Y).
```

This rule is transformed as follows.

```
subclass(⟨X, J⟩, ⟨X, J⟩).
subclass(⟨X, J⟩, ⟨Y, K⟩) :-
  extends(⟨X, J⟩, ⟨Z, L⟩),
  subclass(⟨Z, L⟩, ⟨Y, K⟩).
```

A list of all the reformulated rules is available in a technical report [6, Figure 2].

4.5 Commitment Assertion

Suppose that a classfile with classname X is being verified remotely, and that X extends a class with name Y. The verifier must assert a commitment specifying this subclassing relationship. However, at remote-certification time, the defining classloaders J and K for the classes X and Y respectively are unknown, so the commitment cannot be phrased in terms of $\langle X, J\rangle$ and $\langle Y, K\rangle$. Three reformulations address this problem. First, the remote verification procedure instead formulates the commitment:

```
extends(this, Y)
```

The relative reference `this` represents the class being verified. When the commitments are actually asserted into the commitment database by the local "**verify** $\langle X, J\rangle$" primitive, the defining classloader J for class X is known. Therefore, whenever "**verify** $\langle X, J\rangle$" asserts a commitment p, it *systematically* tags the commitment as p @ $\langle X, J\rangle$. For example, the commitment above will be asserted as:

```
extends(this, Y) @ ⟨X, J⟩
```

A list of all the reformulated commitments that a remote bytecode verifier should generate can be found in the technical report [6, Figure 4].

Second, execution of the **bind** primitive contributes binding information by asserting commitments. Whenever a "**bind** X^L **to** $\langle X, J\rangle$" primitive terminates, it asserts the commitment "$X^L \mapsto \langle X, J\rangle$". These facts will be used for explicit resolution of symbols in commitments and queries.

Third, the initial theory is augmented with translation rules that express how subgoals expressed in terms of loaded class notations may be satisfied using corresponding goals using tagged commitments. For example, to evaluate queries of the form `subclass(`$\langle X, J\rangle$`,` $\langle Y, K\rangle$`)`, we will eventually need to evaluate subgoals of the form `extends(`$\langle X, J\rangle$`,` $\langle Y, K\rangle$`)` using the tagged commitments above. To do so, the following translation rule is used.

```
extends(⟨X, J⟩, ⟨Y, K⟩) :-
  extends(this, Y) @ ⟨X, J⟩,
  Y^J ↦ ⟨Y, K⟩.
```

The rule basically retrieves the corresponding tagged commitment, and then validates binding information by consulting the binding commitments. A similar translation rule is required for each predicate that may be asserted as a commitment. The formulation of these rules is straightforward and the complete set is presented in the technical report [6, Figure 6].

4.6 Obligation Attachment

As with commitments, a remote verifier cannot identify the defining classloader for the classes appearing in obligations. We then follow the same strategy and formulate obligations in terms of static classnames, and then rely on the local **verify** primitive to tag the obligations with the context in which they are to be evaluated. For example, the remote verifier may formulate an obligation of the following form:

$$\texttt{subclass}(Y, Z)$$

When the local **verify** primitive processes this obligation, it tags the query with an evaluation context before attachment:

$$\texttt{subclass}(Y, Z) \,@\, \langle X, J\rangle$$

Similar tagging is systematically applied to each obligation [6, Figure 7].

Translation rules transform tagged queries into queries in terms of loaded classes. For example, the following rule is required in the initial theory in order to handle all `subclass/2` queries:

$$\begin{array}{l}\texttt{subclass}(Y, Z) \,@\, \langle X, J\rangle \;\texttt{:-}\\ \quad Y^J \mapsto \langle Y, K\rangle,\\ \quad Z^J \mapsto \langle Z, L\rangle,\\ \quad \texttt{subclass}(\langle Y, K\rangle, \langle Z, L\rangle).\end{array}$$

The translation rule basically resolves all the symbols in the tagged context, and evaluates a corresponding query in terms of loaded classes. The complete set of these translation rules is presented in the technical report [6, Figure 8].

Recall that, in the running example, the above subclassing obligation should be attached to the primitive "**resolve** $Z{::}M(S)$ **in** $\langle X, J\rangle$", which is identified by the loaded class reference $\langle X, J\rangle$. The remote verifier cannot completely identify the target primitives to which the obligation is attached. Fortunately, obligations are always attached to primitives that are operating on the class being verified [6, Figure 7]. The remote verifier may thus formulate the target of attachment in terms of place holders:

resolve $Z{::}M(S)$ **in** $\_\_$

and rely on the local "**verify** $\langle X, J\rangle$" primitive to fill in $\langle X, J\rangle$, as it does when tagging obligations.

4.7 Linking Strategy

A linking strategy schedules the execution of linking primitives, and coordinates incremental proof linking. The following notations are used to express ordering constraints. For linking primitives p and q, the constraint "$p < q$" requires that any execution of primitive q should be preceded by the completion of primitive p. The constraint "$p < q$ if g" requires that execution of q must not begin if g holds and p has not yet completed.

Java proof linking requires the ordering constraints shown in Figure 2. Except for the newly introduced *Proper Resolution Property [PR]*, these constraints for the multiple-classloader case are extensions to the corresponding constraints for the single classloader case [7, Sec. 4.1].

4.8 Putting It All Together

To illustrate how the scheme above works, consider a refinement of the running example. Suppose class $\langle A, L_1\rangle$ defines a method $M(S)$. Suppose further that $\langle A, L_1\rangle$ has a direct subclass $\langle B, L_2\rangle$, which in turn has a direct subclass $\langle C, L_3\rangle$. Assume that $\langle C, L_3\rangle$ overrides the method $M(S)$. Say the loaded method $\langle C{::}M(S), L_3\rangle$ contains an `invokespecial` instruction that delegates the call to $\langle A{::}M(S), L_1\rangle$. The obligation $\texttt{subclass}(C, A) \,@\, \langle C, L_3\rangle$ will be attached to the primitive "**resolve** $A{::}M(S)$ **in** $\langle C, L_3\rangle$". When the obligation is checked, the subgoals in Figure 3 will be generated. The original obligation

[NP] **Natural Progression Property:** The natural life cycle of a class $\langle X, J\rangle$ is reflected in the ordering below:

$$\begin{aligned}\textbf{load}\ \langle X,J\rangle < \textbf{verify}\ \langle X,J\rangle < \textbf{bind}\ X^J\ \textbf{to}\ \langle X,J\rangle \\ < \textbf{endorse}\ \langle X,J\rangle < \textbf{resolve}\ Y\ \textbf{in}\ \langle X,J\rangle < \textbf{resolve}\ Y{::}M(S)\ \textbf{in}\ \langle X,J\rangle\end{aligned}$$

[PR] **Proper Resolution Property:** The defining classloader of a loaded class is used for resolving the symbolic references of the class:

$$\textbf{bind}\ Y^J\ \textbf{to}\ \langle Y,K\rangle < \textbf{resolve}\ Y\ \textbf{in}\ \langle X,J\rangle$$

Delegation of classloading bottoms out when a classloader defines the requested class:

$$\textbf{bind}\ Y^K\ \textbf{to}\ \langle Y,K\rangle < \textbf{bind}\ Y^J\ \textbf{to}\ \langle Y,K\rangle$$

[IC] **Import-Checked Property:** Resolving a symbolic reference requires that the target object is well-defined:

$$\begin{aligned}\textbf{endorse}\ \langle Y,K\rangle < \textbf{resolve}\ Y\ \textbf{in}\ \langle X,J\rangle \quad & \text{if } Y^J \mapsto \langle Y,K\rangle \\ \textbf{endorse}\ \langle Y{::}M(S),K\rangle < \textbf{resolve}\ Y{::}M(S)\ \textbf{in}\ \langle X,J\rangle \quad & \text{if } Y^J \mapsto \langle Y,K\rangle\end{aligned}$$

[SD] **Subtype Dependency Property:** To establish an obligation concerning a class, type information about its superclasses and superinterfaces might be needed. For example, to establish that the direct superclass Y^J of a loaded class $\langle X, J\rangle$ is subclassable (i.e. `subclassable(`Y`)` @ $\langle X, J\rangle$), We require that all superclasses and superinterfaces of $\langle X, J\rangle$ to be loaded, verified and bound before $\langle X, J\rangle$ is used. To address this need, we require that

$$\textbf{bind}\ Y^L\ \textbf{to}\ \langle Y,K\rangle < \textbf{endorse}\ \langle X,J\rangle \quad \text{if } \texttt{subtypedependent}(Y^L)\ @\ \langle X,J\rangle$$

where the conditional query is handled by the following rules in the initial theory:

```
subtypedependent(X^J) @ ⟨X, J⟩.
subtypedependent(Y^L) @ ⟨X, J⟩ :-
    subtypedependent(Z^K) @ ⟨X, J⟩,
    Z^K ↦ ⟨Z, L⟩,
    extends(this, Y) @ ⟨Z, L⟩.
```

```
subtypedependent(Y^L) @ ⟨X, J⟩ :-
    subtypedependent(Z^K) @ ⟨X, J⟩,
    Z^K ↦ ⟨Z, L⟩,
    implements(this, I) @ ⟨Z, L⟩,
    member(Y, I).
```

[RD] **Referential Dependency Property:** Sometimes, verification of a class Y is needed before we can safely endorse a method $\langle X{::}M(S), J\rangle$. For example, if method $\langle X{::}M(S), J\rangle$ assigns a reference of type Y to a variable of type Z, then Java type rules require Z to be either a superclass or a superinterface of Y. Unless Y is a superclass of X, it is entirely possible that the superclasses and superinterfaces of Y are not verified yet. Consequently, the required supporting commitments for the obligation are not necessarily present at the time the obligation is checked, a violation of the Completion Condition. In such a case, we say that Y is *relevant* to the endorsing of $\langle X{::}M(S), J\rangle$. We assume that, remote verification of the bytecode for method $\langle X{::}M(S), J\rangle$ generates commitments `relevant(`Y`, this::`$M(S)$`)` @ $\langle X, J\rangle$ for all relevant class symbols Y, and we require that:

$$\textbf{endorse}\ \langle Y,K\rangle < \textbf{endorse}\ \langle X{::}M(S),J\rangle \quad \text{if } \texttt{relevant}(Y, \texttt{this::}M(S))\ @\ \langle X,J\rangle$$

That is, we want to collect the commitments for all relevant classes (plus their superclasses and superinterfaces) before we check the obligations attached to "**endorse** $\langle X{::}M(S), J\rangle$".

Figure 2: The Extended Java Linking Strategy

1. subclass(C, A) @ $\langle C, L_3\rangle$	/* **resolve** $A::M(S)$ **in** $\langle C, L_3\rangle$ */
1.1. $C^{L_3} \mapsto \langle C, L_3\rangle$	/* **bind** C^{L_3} **to** $\langle C, L_3\rangle$ */
1.2. $A^{L_3} \mapsto \langle A, L_1\rangle$	/* **bind** A^{L_3} **to** $\langle A, L_1\rangle$ */
1.3. subclass$(\langle C, L_3\rangle, \langle A, L_1\rangle)$	
1.3.1. extends$(\langle C, L_3\rangle, \langle B, L_2\rangle)$	
1.3.1.1. extends(this, B) @ $\langle C, L_3\rangle$	/* **verify** $\langle B, L_2\rangle$ */
1.3.1.2. $B^{L_3} \mapsto \langle B, L_2\rangle$	/* **bind** B^{L_3} **to** $\langle B, L_2\rangle$ */
1.3.2. subclass$(\langle B, L_2\rangle, \langle A, L_1\rangle)$	
1.3.2.1. extends$(\langle B, L_2\rangle, \langle A, L_1\rangle)$	
1.3.2.1.1. extends(this, A) @ $\langle B, L_2\rangle$	/* **verify** $\langle B, L_2\rangle$ */
1.3.2.1.2. $A^{L_2} \mapsto \langle A, L_1\rangle$	/* **bind** A^{L_2} **to** $\langle A, L_1\rangle$ */
1.3.2.2. subclass$(\langle A, L_1\rangle, \langle A, L_1\rangle)$	

Figure 3: Subgoals generated by evaluating subclass(C, A) @ $\langle C, L_3\rangle$

is shown as the top-level goal, annotated with "**resolve** $A::M(S)$ **in** $\langle C, L_3\rangle$", the primitive to which the obligation is attached. We have also annotated all the innermost subgoals with the primitives that assert their matching commitments.

The deduction is successful because the commitments required by the innermost subgoals are already asserted at the time the obligation is checked, that is, at the time "**resolve** $A::M(S)$ **in** $\langle C, L_3\rangle$" is executed. For example, subgoal 1.1 is satisfiable because, according to the Natural Progression Property *[NP]*, the primitive "**bind** C^{L_3} **to** $\langle C, L_3\rangle$" has already been executed. Also, subgoal 1.2 is satisfiable because

bind A^{L_3} **to** $\langle A, L_1\rangle$
$<$ **resolve** A **in** $\langle C, L_3\rangle$ ...*[PR]*
$<$ **resolve** $A::M(S)$ **in** $\langle C, L_3\rangle$...*[NP]*

The rest of the subgoals are more interesting. Note that subtypedependent(B^{L_3}) @ $\langle C, L_3\rangle$ is satisfiable before "**resolve** $A::M(S)$ **in** $\langle C, L_3\rangle$" is executed. By applying the Subtype Dependency Property *[SD]* and other ordering constraints, we deduce

verify $\langle B, L_2\rangle$
$<$ **bind** B^{L_2} **to** $\langle B, L_2\rangle$ ...*[NP]*
$<$ **bind** B^{L_3} **to** $\langle B, L_2\rangle$ ...*[PR]*
$<$ **endorse** $\langle C, L_3\rangle$ ...*[SD]*
$<$ **resolve** $B::M(S)$ **in** $\langle C, L_3\rangle$...*[NP]*

That is, the commitments extends(this, B) @ $\langle C, L_3\rangle$ and $B^{L_3} \mapsto \langle B, L_2\rangle$ (generated by "**verify** $\langle B, L_2\rangle$" and "**bind** B^{L_3} **to** $\langle B, L_2\rangle$" respectively) are already in place when the obligation is checked. Therefore, subgoals 1.3.1.1. and 1.3.1.2. are necessarily satisfiable. Similar reasoning applies to subgoals 1.3.2.1.1. and 1.3.2.1.2.

This example is really a skeleton for the proof of Completion, one of the three correctness criteria for proof linking. These criteria are considered in detail in the next section.

5 Correctness

Given a well-defined linking strategy, proof linking is correct if we can establish the three correctness conditions: Safety, Monotonicity and Completion [7, Sec. 3.3].

5.1 Safety and Monotonicity

Establishment of the Safety and Monotonicity properties follows the corresponding arguments for the single-classloader case [7, Sec. 4.3].

1. **Safety:** Notice that, when the local "**verify** $\langle C, L_3\rangle$" primitive generates the obligation subclass(C, A) @ $\langle C, L_3\rangle$, the obligation is attached to "**resolve** $A::M(S)$ **in** $\langle C, L_3\rangle$". By the Natural Progression Property *[NP]*, the obligation is always attached on time. Similar reasoning can be applied to all the obligations [6, Figure 5].

(α-0)	$X_0^{L_0} \mapsto \langle X_0, L_0 \rangle$	**bind** $X_0^{L_0}$ **to** $\langle X_0, L_0 \rangle$
(γ)	$X_n^{L_0} \mapsto \langle X_n, L_n \rangle$	**bind** $X_n^{L_0}$ **to** $\langle X_n, L_n \rangle$
(β-0)	extends(this, X_1) @ $\langle X_0, L_0 \rangle$	**verify** $\langle X_0, L_0 \rangle$
(α-1)	$X_1^{L_0} \mapsto \langle X_1, L_1 \rangle$	**bind** $X_1^{L_0}$ **to** $\langle X_1, L_1 \rangle$
(β-1)	extends(this, X_2) @ $\langle X_1, L_1 \rangle$	**verify** $\langle X_1, L_1 \rangle$
(α-2)	$X_2^{L_1} \mapsto \langle X_2, L_2 \rangle$	**bind** $X_2^{L_1}$ **to** $\langle X_2, L_2 \rangle$
(β-2)	extends(this, X_3) @ $\langle X_2, L_2 \rangle$	**verify** $\langle X_2, L_2 \rangle$
(α-3)	$X_3^{L_2} \mapsto \langle X_3, L_3 \rangle$	**bind** $X_3^{L_2}$ **to** $\langle X_3, L_3 \rangle$
$\vdots$	$\vdots$	$\vdots$
(β-($n-1$))	extends(this, X_n) @ $\langle X_{n-1}, L_{n-1} \rangle$	**verify** $\langle X_{n-1}, L_{n-1} \rangle$
(α-n)	$X_n^{L_{n-1}} \mapsto \langle X_n, L_n \rangle$	**bind** $X_n^{L_{n-1}}$ **to** $\langle X_n, L_n \rangle$

Figure 4: Leaves of the Proof Tree for the Obligation subclass(X_0, X_n) @ $\langle X_0, L_0 \rangle$

2. **Monotonicity:** The initial theory, commitments and obligations forms a monotonic, horn clause logic (see [6] for details).

5.2 Completion

Completion has to be established on an obligation-by-obligation basis. Continuing with our running example in section 4.8, we consider an obligation subclass(X_0, X_n) @ $\langle X_0, L_0 \rangle$ that is attached to the primitive "**resolve** X_n::$M(S)$ **in** $\langle X_0, L_0 \rangle$". Our goal is to show that, if the predicate subclass(X_0, X_n) @ $\langle X_0, L_0 \rangle$ eventually becomes provable, then it is necessarily provable before the primitive "**resolve** X_k::$M(S)$ **in** $\langle X_0, L_0 \rangle$" is executed.

Suppose that the obligation subclass(X_0, X_n) @ $\langle X_0, L_0 \rangle$ becomes provable at a certain point. Generalizing the proof found in Figure 3, the proof tree of the obligation contains the innermost subgoals in Figure 4.

We number the subgoals as (α-i), (β-i) and (γ). We want to show that the primitives that assert commitments satisfying these subgoals have all been executed prior to the execution of "**resolve** X_n::$M(S)$ **in** $\langle X_0, L_0 \rangle$". As already explained in Section 4.8, the Proper Resolution Property *[PR]* guarantees that supporting commitment (γ) is already in place. We use induction to show that commitments (α-i) and (β-i) are already asserted when the obligation is checked.

Basis: Commitment (α-0) and (β-0) are already asserted because,

verify $\langle X_0, L_0 \rangle$
$<$ **bind** $X_0^{L_0}$ **to** $\langle X_0, L_0 \rangle$ ...*[NP]*
$<$ **resolve** X_k::$M(S)$ **in** $\langle X_0, L_0 \rangle$...*[NP]*

Induction Step: Assume that commitments (α-i) and (β-i) are already in place, for $0 \leq i < k$, where $k > 0$. The presence of these commitments enable the query subtypedependent($X_k^{L_{k-1}}$) @ $\langle X_0, L_0 \rangle$ to be satisfiable. We then have

verify $\langle X_k, L_k \rangle$
$<$ **bind** $X_k^{L_k}$ **to** $\langle X_k, L_k \rangle$ ...*[NP]*
$<$ **bind** $X_k^{L_{k-1}}$ **to** $\langle X_k, L_k \rangle$ ...*[PR]*
$<$ **endorse** $\langle X_0, L_0 \rangle$ ...*[SD]*
$<$ **resolve** X_n::$M(S)$ **in** $\langle X_0, L_0 \rangle$...*[NP]*

Since the contributors "**bind** $X_k^{L_{k-1}}$ **to** $\langle X_k, L_k \rangle$" and "**verify** $\langle X_k, L_k \rangle$" for respectively (α-k) and (β-k) are already executed, the commitments are present when the obligation is checked.

This concludes the proof of Completion for one class of obligations. Completion can be established similarly for the rest of the obligations.

6 Discussion

6.1 Implementation Guidelines

The linking strategy above suggests a natural implementation of proof linking in a JVM with multiple classloaders. In particular, a call to the `defineClass` method of class loader J, with argument X as the expected classname, will execute "**load** $\langle X, J\rangle$" and "**verify** $\langle X, J\rangle$". A call to the `loadClass` method of class loader K, requesting the loading of class X, corresponds to the primitive "**bind** X^K **to** $\langle X, J\rangle$". The primitive "**endorse** $\langle X, J\rangle$" could be executed when the class "$\langle X, J\rangle$" is prepared [11, Sec. 5.4.2]. The primitive "**endorse** $\langle X::M(S), J\rangle$" could be executed right before the method is first resolved. The resolution primitives coincide with regular symbol resolution.

The translation rules, binding commitments, and tagging can be optimized readily. For example, a classloader usually has a hash table storing the classes whose loading it has initiated. Such a table can be reused to represent binding commitments. Also, tagging is just an abstract way to say that the tagged commitments/obligations are stored in the Class objects themselves. Notice that this suggests a very convenient way to retract commitments/obligations when a class is finalized.

6.2 Comparison with Sun's Linking Strategy

As opposed to Sun's JVM implementation, which postpones bytecode verification until a class is linked, the implementation strategy above performs eager verification, that is, the local "**verify** $\langle X, J\rangle$" primitive is executed immediately after $\langle X, L\rangle$ is defined. This is necessary to ensure that commitments are gathered as soon as possible. Sun's JVM performs one pass of verification at class definition time, postponing the second and third passes until link time.

6.3 Implementation Status

We are in the process of implementing a proof linker in a CLDC-compliant KVM extended with multiple classloaders. The exercise has clarified a lot of our thoughts, and has confirmed the compatibility of proof linking and Java's delegation style classloading mechanism.

7 Conclusion

We advocate the adoption of the proof linking architecture as a standard framework for conducting distributed verification for the JVM. The architecture supports the notion of conditional certification essential for signature-based verification protocols, and offers interoperability among various distributed verification protocols. We have also extended our original proof linking model to account for the presence of multiple classloaders in the standard JVM, thereby showing that proof linking is applicable to complex mobile code environments such as J2SE.

Acknowledgement

The research was funded in part by a scholarship and an operating grant from the Natural Sciences and Engineering Research Council of Canada.

References

[1] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, Vancouver, BC, Canada, June 2000. Also available as `http://www-nt.cs.berkeley.edu/home/necula/public_html/pldi00b.ps.gz`.

[2] Prem Devanbu and Stuart Stubblebine. Automated software verification with trusted hardware. In *Proceedings of the Twelfth International Conference on Automated Software Engineering*, 1997.

[3] Premkumar T. Devanbu, Philip W. L. Fong, and Stuart G. Stubblebine. Techniques for trusted software engineering. In

Proceedings of the 20th International Conference on Software Engineering (ICSE'98), Kyoto, Japan, April 1998. Also available at `http://seclab.cs.ucdavis.edu/~devanbu/files/icse98.pdf`.

[4] Federal Information Processing Standards Publication. Security requirements for cryptographic modules. Technical Report FIPS PUB 140-1, U.S. Department of Commerce/National Institute of Standards and Technology, January 1994. Available as `http://csrc.nist.gov/fips/fips1401.pdf`.

[5] Philip W. L. Fong and Robert D. Cameron. Proof linking: An architecture for modular verification of dynamically-linked mobile code. In *Proceedings of the Sixth ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'98)*, Orlando, Florida, USA, November 1998. Also available at `http://www.cs.sfu.ca/~pwfong/personal/Pub/fse98.ps`.

[6] Philip W. L. Fong and Robert D. Cameron. Java proof linking with multiple classloaders. Technical Report SFU CMPT TR 2000-04, Simon Fraser University, 2000. Available at `ftp:// fas.sfu.ca/pub/cs/TR/2000/`.

[7] Philip W. L. Fong and Robert D. Cameron. Proof linking: Modular verification of mobile programs in the presence of lazy, dynamic linking. *ACM Transactions on Software Engineering and Methodology*, 2000. To appear. Also available as `http://www.cs.sfu.ca/~pwfong/personal/Pub/tosem2000.ps`.

[8] IBM. IBM PCI cryptographic coprocessor. Available at `http://www-3.ibm.com/security/cryptocards`.

[9] Gerwin Klein and Toblias Nipkow. Verified lightweight bytecode verification. In *ECOOP 2000 Workshop on Formal Techniques for Java Programs*, 2000. Also available at `http://www4.informatik.tu-muenchen.de/~nipkow/pubs/lbv.ps`.

[10] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)*, pages 36–44, Vancouver, British Columbia, October 1998. Also available at `http://java.sun.com/people/gbracha/classloader.ps`.

[11] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 2nd edition, 1999. Also available at `http://java.sun.com/docs/books/vmspec`.

[12] Gary McGraw and Edward W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes.* Wiley, 1997.

[13] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, Paris, France, January 1997. Also available at `http://www-nt.cs.berkeley.edu/home/necula/public_html/popl97.ps.gz`.

[14] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI'96)*, Seattle, WA., October 1996. Also available at `http://www.cs.cmu.edu/~necula/osdi96.ps.gz`.

[15] Eva Rose and Kristoffer Hogsbro Rose. Lightweight bytecode verification. In *The OOPSLA'98 Workshop on Formal Underpinnings of Java*, Vancouver, BC, Canada, November 1998. Available at `http://www-dse.doc.ic.ac.uk/~sue/oopsla/rose.f.ps`.

[16] Sun Microsystems. *Connected, Limited Device Configuration: Java 2 Platform Micro Edition.* Sun Microsystems, version 1.0 edition, May 2000. Available at `http://java.sun.com/products/cldc/`.

[17] WAP Forum. *WAP-193-WMLScript Language Specification.* WAP Forum, version 1.2 edition, June 2000. Available at `http://www.wapforum.org/what/technical.htm`.

JVM Susceptibility to Memory Errors

Deqing Chen[†], Alan Messer, Philippe Bernadat, Guangrui Fu,
Zoran Dimitrijevic[††], David Jeun Fung Lie[‡‡], Durga Mannaru[*], Alma Riska[‡], and Dejan Milojicic

Univ. of Rochester[†], HP Labs, UCSB[††], Stanford Univ.[‡‡], Georgia Tech.[*], William and Mary College[‡]

lukechen@cs.rochester.edu[†], [messer, bernadat, guangrui, dejan]@hpl.hp.com,
zoran@cs.ucsb.edu[††], davidlie@stanford.edu[‡‡], durga@cc.gatech.edu[*], riska@cs.wm.edu[‡]

Abstract

Modern computer systems are becoming more powerful and are using larger memories. However, except for very high end systems, little attention is being paid to high availability. This is particularly true for transient memory errors, which typically cause the entire system to fail. We believe that this situation can be improved by addressing memory errors at all levels of the system, bring commodity systems closer to mainframe-class availability.

In this paper, we use fault injection experiments to investigate memory error susceptibility at the highest level using a JVM and four Java benchmark applications. We then consider JVM data structure checksums to increase detection of silent data corruption affecting the JVM and applications. Our results indicate that the JVM's heap area has a higher memory error susceptibility than its static data area and that we can detect up to 39% of all memory errors in the JVM and application. We believe that such techniques will allow commodity systems to be made much more robust and less error-prone to transient errors.

1 Introduction

The demand for high performance and availability in commodity computers is increasing with the ubiquitous use of computers and Internet services. While commodity systems are tackling the performance issues, availability has received less attention. It is a common belief that software errors and administration down-time are, and will continue to be, the most probable cause of loss of availability. While such failures are clearly commonplace, especially in desktop environments, the probability of certain hardware errors is increasing.

Hardware errors can be classified as hard errors and transient (soft) errors. Hard errors are those that require replacement (or otherwise relinquished use) of the component. They are typically caused by physical damage to a component, e.g. by damage to connectors. Transient errors are those that result in an invalid state that can be corrected, for example, by overwriting a corrupt memory location. Ziegler et al. [21, 22] have shown that factors such as increased semiconductor technology density and reduced supply voltage will lead to increased transient errors in CMOS memory because of the effects of cosmic rays. Tandem [19] indicates that such errors also apply to processor cores and on-chip caches at modern die sizes andvoltage levels.

Although the increased use of Error Correction Codes (ECC) can significantly reduce the probability of these transient errors, greater speeds, denser technology, and lower voltages increase the likelihood of these errors becoming significant in future systems. Even if ECC protection is used, multiple bit errors may still escape the scope of the hardware protection and corrupt values in random memory locations. Applications can then potentially use incorrect value on their next access, this is called "silent data corruption." Typical examples are transient errors in the processor registers, in the ALU, multiple-bit memory errors, and so forth. As a result, when these errors escape hardware protection, it is only possible for software to detect them.

In some of the most promising applications of Java technologies, such as in embedded systems, no parity or ECC protection is used, allowing more of these errors to be exposed to the system. In current commodity systems, there is little consideration for transient memory errors. For example, in most systems based on the IA-32 architecture [9], when a transient memory error occurs, the CPU simply enters a Machine Check Abort (MCA) exception from which the OS can only panic or reboot.

However, in the new IA-64 architecture [8], there is increased scope for useful MCA handling. At the time of the MCA exception, the CPU can provide much more information about the current CPU status and can notify the operating system to handle the exception. This ability provides new opportunities for future systems to recover more gracefully from memory errors.

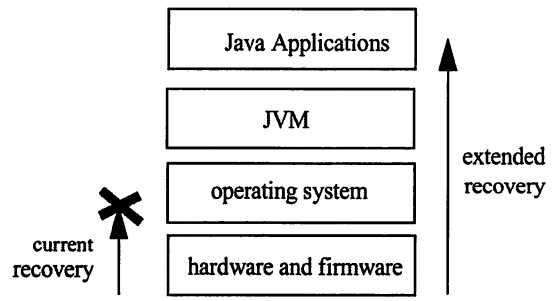


Figure 1 Propagating Memory Errors. *Memory errors are detected by lower layers and either corrected or propagated to higher levels of the system, up to applications*

Existing research [12] has outlined the opportunity for memory error recovery with increased hardware support. This research proposes that the operating system can be extended to increase recoverability when it receives a memory error exception. However, recoverability of the whole system is complex and involves participation at all levels from the hardware to the application software. We propose that if the OS determines that a memory error occurred in an application, it can deliver the error exception to the application for further processing. In this paper we focus on Java Virtual Machines (JVM) and Java applications for exception handling at this level (see Figure 1).

At the application level, JVMs and Java applications are of particular interest because of their large garbage-collected heaps, the virtual machine abstraction presented, and the integrated exception mechanism. Large garbage-collected heaps present a sweet-spot for this research, because the garbage collector itself may uncover many errors as part of the heap sweep during collection. These heaps are also usually larger than explicitly allocated heaps, thereby increasing the probability of a memory error during a sweep.

By presenting an abstraction between the operating system and the applications, the virtual machine makes application-level recovery simpler. Since, the JVM has increased information about the application's status and semantics, such as memory usage, there is an improved chance of recovery.

Java's integrated exception handling could allow applications to be written that are memory error aware [12] by trapping new exceptions. If the virtual machine can isolate the error solely to the application, it can generate these exceptions and allow the application to handle the memory error gracefully.

However, memory failure recoverability is a complex problem. This paper tries to identify the memory error susceptibility in the Java virtual machine and Java applications as a first step towards tackling this potential problem. The major contributions in this paper include: quantifying the memory error consumption and susceptibility rate in the Kaffe JVM and sample Java applications; and, evaluation of extensions to the Kaffe JVM to detect silent data corruption.

The rest of the paper is organized as follows. In Section 2, the paper outlines work related to the problem. Section 3 describes the problems that we are addressing. The methodology of the fault injection experiment and the method for detecting silent data corruption are described in Section 4. Section 5 presents the experimental results. Lessons learned are presented in Section 6. The paper ends with recommendations for future work in Section 7 and conclusions for this work in Section 8.

2 Related Work

The effects of and trends for soft-errors were first reported by Ziegler et al. [21, 22], based on field and experimental evidence that alpha particles and cosmic rays were the source of several random system failures. Since then, soft errors have become a greater concern because semiconductor susceptibility to these particles increases with technology density and voltage drops.

Availability in computer systems is determined by hardware and software reliability. A high level of hardware reliability has traditionally existed only in proprietary servers, with specialized, redundantly configured hardware and critical software components, possibly with support for processor pairs [2], e.g., IBM's S/390 Parallel Sysplex [15] and Tandem's NonStop Himalaya [5].

Reliability has been more difficult to achieve in commodity software even with extensive testing and quality assurance [13, 14]. Commodity software fault recovery has not evolved too far at this time. Most operating systems support some form of memory protection between units of execution to detect and prevent wild read/writes. But most commodity operating systems have not addressed problems of memory errors themselves or taken up software reliability research in general. Examples include Windows 2000 and Linux. They typically rely on fail-over solutions, such as Wolfpack by Microsoft [16] and High-Availability Linux projects [20].

A lot of work has been undertaken in the fault-tolerant community regarding the problem of reliability and software recovery [3, 7, 11]. These include techniques such as check-pointing [7] and backward error recovery [3]. Much of this work has been conducted in the context of distributed systems rather than in single systems. There are also techniques for efficient recoverable software components, e.g., RIO file cache [4] and Recoverable Virtual Memory (RVM) [17].

The Fine [10] project uses fault injection techniques to study the fault tolerance of UNIX systems. Fine is a set of experimental tools capable of injecting hardware- and software-induced errors into the UNIX kernel and tracing the execution flow and kernel's key variables. Our fault injection work operates at the application level and uses the debugger tool *ptrace* to trace the application's behavior.

Some research has attempted to quantify the absolute number of errors that would be seen in particular configurations [21, 19, 6]. For example, it is estimated that a 1Gb memory system based on 64Mbit DRAMs still has a combined visible error rate of 3435 Failures In Time (FIT – errors in one billion hours) when using Single Error Correct-Double Error Detect (SEC-DED) ECC [6]. This is equivalent to around 900 errors in 10,000 machines in 3 years. Tandem [19] estimates that a typical processor's silicon can have a soft-error rate of 4,000 FIT, of which approximately 50% will affect processor logic and 50% will affect the large on-chip cache. Due to increasing speeds, denser technology, and lower voltages, such errors are likely to become more probable than other single hardware component failures.

Most recently, HP Labs has studied the future trends of these error rates, their repercussions on processor error handling support, operating system handling/recovery, and application recoverability [12]. This paper reports part of this.

3 Memory Error Susceptibility

Memory errors present themselves in a computer system as either serious exceptions, when detected, or silent data corruption in memory, if undetected. However, in many current Java environments, memory errors will be discovered as silent data corruption since no memory detection or correction hardware is used. In this paper, we concentrate on the analysis and recovery of those corruptions that occur in the application's data area. Errors in the native instruction sequence and errors in the kernel area are beyond the scope of this study and are addressed elsewhere [12].

Suppose a transient error happens on a word inside an application's data area, the error may or may not be consumed (accessed) by the application. If the error is consumed, the error may or may not eventually lead to an application error. For example, suppose an error occurs on an ID string array so that one ID is changed unexpectedly. If this ID is never matched in searches, the error won't lead to any application errors.

Studying the affect of transient memory errors on JVMs and Java applications has many valuable benefits. Most of all, it lets us understand the application behavior under silent data corruption so that we can design efficient software methods to detect silent data corruption. Since it's infeasible to detect all of the errors, our study focuses on data areas most susceptible to memory errors. The rest of this section defines the terms we used in the paper and describes the experimental environment used.

3.1 Memory Error Definitions

We refer to the act of an application accessing a memory location containing a soft error as *error consumption*. We define the memory error consumption rate ($R_{consumption\_rate}$) as the ratio of the number of errors consumed ($N_{error\_consumed}$) versus the number of memory errors (N_{memory_errors}), i.e.,

$$R_{consumption\_rate} = N_{error\_consumed} / N_{memory\_errors}$$

This equates to the portion of the total error rate that is actually seen by the application, because only errors in those memory locations that are accessed are noticed. The consumption rate is always smaller than one. Thus, our definition of consumption rate is the upper bound on errors seen by the execution in a real situation. For simplification, in this paper, we assume a memory error persists until it is consumed or the application exits. This is necessary because some high-end operating systems use a memory scrubber to pass over physical memory removing any correctable errors it finds. In the presence of ECC memory, the memory scrubber can clear all correctable errors that exist in memory.

If the error consumption eventually causes the application to crash or to return an erroneous result, we say that it has caused an *application error*. Verification of the latter is performed by comparing the result against a known correct result. Lastly, we refer to the *error susceptibility* of a memory region as the likelihood of an application error being caused on error consumption. The memory susceptibility ($S_{susceptibility}$) for a memory

area is defined as the ratio of actual application errors ($N_{errors\_in\_application}$) divided by the number of memory errors (as in the previous formula), i.e.,

$$S_{susceptibility} = N_{errors\_in\_application} / N_{memory\_errors}$$

We assume that memory errors are distributed uniformly in the application's total virtual memory area. Since memory errors affect physical memory, this is similar to assuming that the working set fits into physical memory.

3.2 JVM Memory Error Susceptibility

In a JVM, the data area can be divided roughly into two partitions, those allocated statically for the virtual machine (VM) and those allocated on the heap for Java objects. We want to identify the error susceptibility of these two different memory areas to guide future recovery studies. For errors in the heap, we also want to know how the susceptibility varies with different heap object types.

One feature of the JVM is that unused Java objects are not freed explicitly by the application; rather, they are collected and freed by the garbage collector. How the garbage collector (GC) consumes memory errors is also interesting.

Since all silent data corruption is not detected by hardware solutions, we need to design a software solution to detect these errors. We propose a simple detection scheme using checksumming of heap objects. Fault injection will be used to evalulate the efficiency of this approach.

3.3 Experimental Setup

We chose Kaffe for experimentation because it is an open source package that allows us to get its source code and extend it freely. Having its source code allows us to examine its memory usage, to instrument it for fault injection experiments, and to extend it to detect silent data corruption. It is also a mature system, has reasonable performance, and is widely used.

For our experiments, we used Redhat Linux 6.2, running Kaffe 1.0.5 with the "interpreter mode." Since we assume an IA-64 error handling architecture and Kaffe has not been ported to IA-64 yet, we used a IA-32 architecture Pentium-III processor based system instead. Where appropriate, we will point out the different memory error implications of using each type of processor.

4 Experiment Methodology

In this section, we first explain the method and setup of the fault injection experiments. Next we describe our prototype implementation for detecting silent data corruptions.

4.1 Fault Injection Experiment Method

Our basic experiment method is to inject errors into the application data area, track the error consumption, and monitor the application behavior after any consumption. We use the *ptrace* system call to trace the JVM execution, and manipulate the debug registers to set a data breakpoint to track the error data consumption.

Data Breakpoints

In the IA-32 architecture, there are eight debugging registers that can be used to set data breakpoints. They are identified as DR0 – DR7. DR6 is the breakpoint status register, DR7 is the debug control register, and DR0 – DR3 are used to set the addresses of breakpoints.

For each breakpoint address, the IA-32 architecture allows the user to set it for breaking on execution, breaking on writes, or breaking on read-write. In this experiment, we set the CPU to break on read-write of the injected-error address. At each time, we set only one address. This method has the limitation that we cannot figure out whether the access is a read or a write. We can overcome this limitation by duplicating the breakpoint and setting one for read-write and the another for write. But we are unable to get the correct debugging status register value from the Linux system. Therefore, we do not know which breakpoint fires. It may be possible to overcome this limitation in the future.

Using ptrace

Debug registers are privileged CPU resources and a user application cannot read and write them directly. Fortunately Linux provides the *ptrace* system call for accessing these registers from user processes.

Normally, a *ptrace* system call is used in the following way. The debug process uses *fork* to create a child process. On return from the fork, the child process calls *ptrace* with the parameter TRACEME to inform the parent process that it wants to be traced. The child process then calls *execl* or other similar functions to execute the debugged application. On the other side, the parent process calls a *wait* on the return from the *fork*. When the

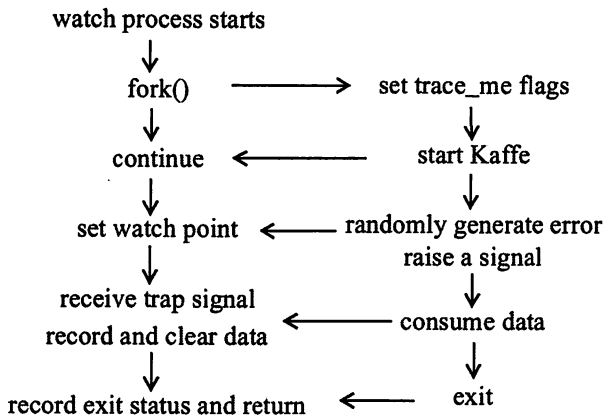


Figure 2 Tracing error consumption using ptrace.

child process first calls *execl*, or generates some uncaught signals, the parent process wakes up from the previous *wait*. After waking the parent process can examine and set the status of the child process with the *ptrace* call.

The way we use *ptrace* is illustrated in Figure 2. We modified the Kaffe executive to start the watch (monitor) process first. The watch process uses *fork* to create and run the VM. At certain points of the VM's execution, a memory error is generated and a SIGTRAP is raised to inform the parent – the watch process – to set a data breakpoint on the error address. On receiving this signal, the watch process peeks at the child process data (because they have the same address space layout, we can obtain the child's data address easily) and sets the appropriate data breakpoint.

After the child process resumes, it may or may not consume the injected error. If the error is consumed, the child process traps and the parent wakes from this trap signal. The consumption is recorded and the breakpoint is cleared. Whenever the child process exits normally or incorrectly, the watch process is signaled and the status is recorded. If the child process exits normally, we further check whether its output is correct.

Generating and Recording Memory Errors

We instrumented the Kaffe virtual machine to inject memory errors into the data memory area and to record the memory status. Since we are using the interpreter mode, the virtual machine executes a loop interpreting each byte code. Code is instrumented so that after a certain number of byte codes have been executed, the loop calls our error injection procedure to generate a memory error.

Each memory error is injected into one of two data memory areas:

- the static memory area of the VM, and
- the object heap.

In each test set, errors are injected into one of the above areas. Each time, a byte is randomly chosen from the specified area and the location's bits are flipped. If the error is injected into the object heap, we record the type information of the object where the byte is located. For our purpose, the information we record includes the object type, size, and base address.

Next, the VM stores the error address into a global variable and raises a SYSTRAP signal to inform the watch process that a memory error has been generated. After receiving this signal, the watch process peeks at the global variable to get the error address and set a data breakpoint at the address. Then the VM is allowed to continue.

When the error is consumed, we also inspect the VM status to see whether it is consumed by the garbage collector. Kaffe uses the mark and sweep algorithm, which makes this inspection fairly easy because when the GC is running all of the other user threads are stopped.

4.2 Detecting Silent Data Corruption

Based on our experimental results on error consumption, we have implemented a prototype solution for detecting silent data corruption for the Kaffe virtual machine. We believe the method can be applied to other virtual machine implementations as well.

The basic idea is that in a pure Java application every Java object or array is accessed through a specific group of bytecode operations, such as `getfield` and `putfield`. For each of these operations, we add code to do a checksum computation. The heap object management can be modified to store the checksum results.

Space For Checksums

Instead of directly extending Kaffe's object data structure to have extra fields for storing checksum data, we extended the heap memory management data structure to have more bytes for each memory block. This conforms to the way that Kaffe manages the object status.

In the Kaffe heap memory management module, objects are classified into small objects and big objects. Small objects are generally objects with sizes smaller than the system page size. Large objects are objects needing more than one page.

Small objects are grouped into pages. Each page is divided into many same-size blocks. Each block is assigned to one object. At the head of the page, there is a meta-data structure for blocks inside the page. It contains information such as block size, garbage collection status, and object type. Two bytes are added for each small object, using one byte for a bit pattern checksum and another for checksum validity. The checksum must be invalidated after native calls because native accesses are not checksummed in our implementation.

For big objects and arrays, it is not efficient to have only one checksum across the whole structure. When one byte in a one-megabyte array is accessed, we do not want to compute a checksum for the whole array. Thus, we divide the object into fixed-size small blocks and the checksum is computed on these small blocks. Although we add extra memory overhead, the checksum is computed much more efficiently for large objects or arrays.

Checksum Computation

When a Java application is running, objects are accessed when:

- it is created using the *new* operator,
- one of its fields is read or written by the bytecodes *get/putfield, get/putstatic*,
- an entry in an array is read by one of the bytecodes: *iaload, laload, faload, daload, caload, saload, baload* and *aaload*,
- an entry in an array is written by one of the bytecodes: *iastore, lastore, fastore, dastore, castore, sastore, bastore* and *aastore*,
- one part of an array is copied by *System.array_copy*,
- the object or array is operated on by some native functions,
- the object is walked by the garbage collector.

In Kaffe, because static fields are class related they are stored within the class objects rather than the data objects. Due to time limitations, we were unable to instrument Kaffe to add checksum protection to the static areas of class objects. Therefore, our results are based only on instrumenting data object accesses.

Using our instrumentation when an object field or an array entry is read by some bytecode, we compute the checksum of the read value with the rest of the object or array and compare it with the checksum we have previously stored in the object's block meta data structure. When an object is updated by a bytecode, we update its checksum value. For simplicity, in our implementation the checksum is computed by XORing all bytes in the object rather than by a polynomial checksum as used in TCP/IP.

5 Experiment Results

In this section, we present our experimental results for error consumption and silent data corruption detection. In our experiments, we assume a uniform memory error probability over the whole memory area. For the convenience of the experiments, we inject the same number of errors in the two experiment sets.

The benchmark applications we used in the experiments are extracted from the SPEC JVM98 benchmark suites [18]. We selected four applications from this suite:

- _202_jess, a Java expert system,
- _209_db, a Java database,
- _213_javac, a Java compiler, and
- _228_jack, a Java parser generator.

In all of the experiments we conducted, we used the medium data configuration – ten percent. With this data size, the experiments finish in a reasonable time, and are large enough to cause the garbage collector to run.

For both static and dynamic areas, we inject 1,000 memory errors for the four benchmarks. For the dynamic area experiments, the benchmarks are run with the error detection mechanism so that we can record which error consumptions have been detected. The total running time for the experiments took about 70 hours on a Pentium III 500MHz platform. The total code size for error injection and tracing is about 470 lines with about 780 lines for memory error detection.

5.1 Memory Error Consumption

This experiment is divided into two parts. In one part, we inject memory errors into the VM's static memory area; in the other part, we inject errors into the object

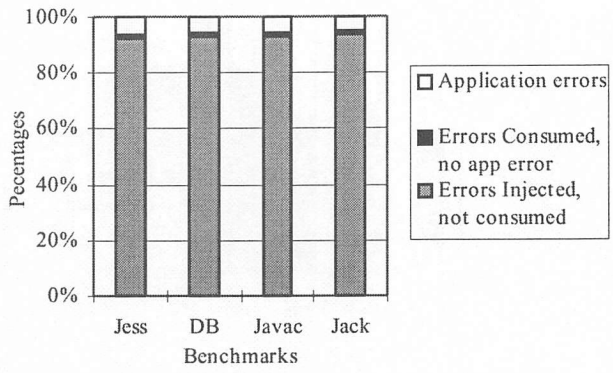


Figure 3 Error consumption in the JVM's static data.

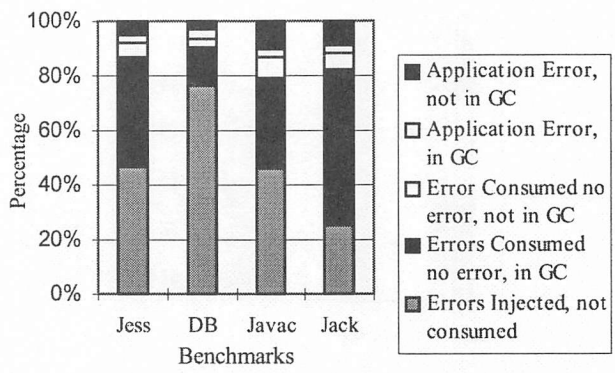


Figure 4 Error Consumption in the JVM's heap region.

heap. These two areas are used differently by Kaffe. The static data area includes the global variables and constants. Intuitively, errors in this area are much more likely to cause real problems in the Java application once they are consumed. On the other hand, a Java application's data objects are stored on the heap which is walked by the garbage collector when it is started. The heap can have a higher error consumption rate than the static data area because of garbage collection.

Static Memory

The results from injecting errors into the static data area are summarized in Figure 3. In the graph, the mid-gray part comprises those errors that are not consumed by the application even though they are injected; the dark-gray part comprises errors that are consumed by the application but don't cause any application errors, i.e., the application accessed the erroneous data but it still executed correctly; the light-gray part illustrates the number of application errors, in this case, the application either crashes or gives a wrong result.

The susceptibility rates are listed in Table 1. The size of this data area is about 350KB. We can see from the graph that all of the benchmark applications exhibit similar behavior. Their error consumption rate is about 6% to 7% with an average of 6.7%. The average memory susceptibility rate is about 5.5%. Among all of the errors consumed, 81% of them cause errors in the applications.

Static Data	Jess	DB	Javac	Jack	Avrg
Susceptibility	6.2%	5.4%	5.4%	5.1%	5.5%

Table 1 Susceptibility in Static Data

Object Heap

In the next experiment, we inject errors into the object heap. In Kaffe, the heap size grows dynamically as the application's need grows. In our experiment, we injected errors into the range of virtual addresses the heap occupies. In these experiments, the application heap sizes varied from 5,243KB to 8,397KB (see Table 2).

Heap Size	Jess	DB	Javac	Jack
Minimum Heap Size	5243KB	7348KB	5243KB	5243KB
Maximum Heap Size	5243KB	8397KB	7000KB	7000KB

Table 2 Heap Size Used in Error Injection

The results from our heap injection experiements are summarized in Figure 4 with the appropriate susceptibility rates listed in Table 3. The three cases (application error, consumed but no error, and injected but not consumed) have the same meaning as in Figure 3.

Object Heap	Jess	DB	Javac	Jack	Avrg
Susceptibility	8.3%	7.1%	13.2%	11.9%	10.1%

Table 3 Susceptibility in the Heap

Our first observation is that the heap has a much higher error consumption rate. For example, Jack has a 75% error consumption rate in the heap versus 6.7% in the static data area. But a closer look reveals that most consumption comes from the garbage collector. Kaffe uses mark and sweep strategies for garbage collection. When collection is started, it touches almost every object in the heap. It is no wonder that it consumes so many errors. If we do not count the errors consumed in the GC, the error consumption rate is about 9% to 22%, which is still higher than in the static data area.

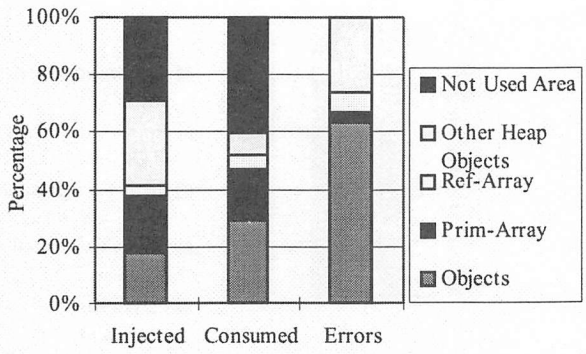


Figure 5 Error Consumption by Object Type.

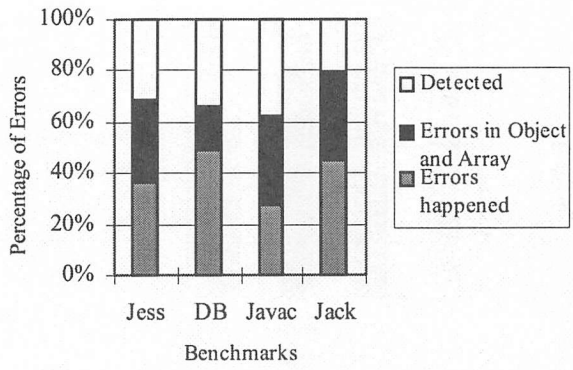


Figure 6 Checksum Detection of Application Errors

It should also be noted that the susceptibility also depends on memory region size. However, if we assume a uniform error probability in the memory area, because the heap size is much bigger than the static area, we can conclude that the heap is still much more susceptible of than the static data.

Although most of the consumption takes place in the garbage collector, relatively few errors actually cause real problems. The first reason is that the garbage collector only cares about an object's reference field. It would not use other types of fields for computations. For an object reference, it first checks whether it is valid, which masks out most of the possible errors. On average, only 7% of the error consumption in the GC caused application errors. In comparison, 56% of static data error consumption caused application errors.

To further understand the source of application errors, we also collect the object types for the object into which each error is injected. In Figure 5, we show the result for Javac. We distinguish objects, primitive arrays, reference arrays, and areas that are not used. An example of the latter, are areas that do not belong to any JVM object, such as an object that has been freed by the garbage collector, or a block inside a page that has not been allocated to any object. These results indicate that errors injected into unused parts never caused application errors. However, they may be consumed by overwriting.

From the graph we can see that although only less than 20% of the errors injected are in normal objects (i.e., objects created with new), they are much more likely to be consumed and cause application errors – more than 60% of application errors are caused by these objects.

We can also see that many errors are injected into primitive arrays. This is understandable because user applications tend to store large data sets in arrays. However, because these are large structures containing particular single errors, these errors are less likely to be consumed because array accesses may rarely use the erroneous data. Therefore, depending on application data usage, errors in primitive arrays may cause less application errors than these error consumption rates indicate. On the other hand, reference arrays are much more likely to cause application errors, because a false pointer can easily cause a segmentation fault in the JVM.

Due to the space limitations, details on other error data types is not included here. Briefly, constant fixed objects occupy a large percentage of the "other heap object" part in Figure 5. These objects include data such as bytecodes and the constant pool. In total, these objects occupy between 8% and 30% of the objects types. Since they are read-only objects recovery of these objects types should be straightforward.

5.2 Checksum Silent Data Corruption Detection

To demonstrate the effectiveness of our scheme for detecting silent data corruption, we implemented a prototype in Kaffe. Compared to the proposal, the prototype implementation has several limitations. First, when native functions or System.array_copy is called, we simply clear the object's or array's checksum validity rather than update the checksum result, although in the future we will do so.

Another limitation is that we do not compute checksums for large objects, although we do deal with large arrays. We assume that we will not see many large objects in Java applications because in a Java object, embedded objects are stored as a reference.

We ran the fault injection experiments on our prototype implementation with the four benchmarks. We recorded the cases when consumed errors are detected. Figure 6 shows the percentage of application errors that can be

detected when the error is consumed. The light-gray areas represent errors detected. The dark-gray areas represent those errors that we know took place in objects and arrays and that we could have corrected if we applied checksumming. It has not been applied because the object is too big or was operated on by some native functions that are not easily checksummed. Finally, the mid-gray area comprises the cases where the memory error was not detected and corrected, and caused an application error.

The effectiveness of the detection depends on the nature of the application. If objects and arrays account for most of the actual errors occurring, the technique is more effective. For example, for Javac, errors in objects and arrays account for nearly 80% of all error occurrences. Our technique can detect up to 39% of all errors in our experiments.

The percentage of errors detected by only the current implementation was limited by time constraints. In the future, the implementation can by improved by updating checksums during native function calls and array copies. The technique can also be extended by including more heap objects into the checksum detection, such as constant pools and bytecode sections. Since these heap objects are never changed after they are loaded, the extra checksumming overhead would be small because only checks on read access would be required.

We also compared the relative slowdown of the prototype implementation with the original Kaffe implementation. It is interesting to see the performance overhead induced by the checksum process. We measured the total execution time of the original Kaffe implementation and our prototype implementation. The relative slowdown compared to the original version is shown in Table 4 for each benchmark used.

	Jess	DB	Javac	Jack
Slowdown	57%	43%	47%	32%

Table 4 VM Slowdown with Detection

6 Lessons Learned

We found that ptrace is a good tool for fault injection experiments. It lets us generate data breakpoints in the Kaffe VM and track the consumption of the injected errors. At the time of error consumption, the breakpoint allows us to stop the VM and examine its internal state. Originally we had thought of collecting execution traces to study the error consumption rate, but it would be extremely difficult for us to derive the VM's status at the time of error consumption from the traces. Of course, ptrace has limitations. It is not clear to us whether we can use it successfully to study kernel mode errors.

From the experiment data and analysis, the following interesting observations can be derived:

- For the Kaffe virtual machine and the Java applications running in it, the memory errors in the object heap have a higher error consumption rate and susceptibility rate than those in the static data area. The heap size is also much larger than the static data size. If we assume a uniform error distribution, we can draw the conclusion that the heap memory will be the dominant part in memory susceptibility.
- A large portion of error consumption in the heap is caused by the garbage collector (up to 75% in the case of Jack). But this consumption leads to less application errors than other consumption (7% vs. 56%).
- For memory errors occurring in the object heap, errors injected in normal objects (created with new) and arrays caused 70% of the application errors.
- By adding simple checksums, normally undetected errors can be detected, increasing error coverage by 30-40%.
- Adding checksums clearly comes at a performance cost. Our unoptimized checksum routine adds this functionality for an increase in run time of 32-57%. Optimizing the checksum computation for the platform (maximizing explicit parallelism) or using hardware support for block checksums should help make this more acceptable for comtemporary JIT run-times.
- The coverage of silent data corruption detection should be easy to increase by placing checksums over more object types (e.g., static objects). The overhead could be further reduced by limiting additional unnecessary checks.
- Several objects in the Java heap can be relatively large and were not covered by our checksums. This assumption should be relaxed for future experimentation.

7 Future Work

Some further work is needed to complete our study of memory failure recoverability at the application level. First, we need to extend and optimize our prototype silent data error corruption implementation to handle other heap objects, including large objects, the constant pool, byte code, etc. Using these extensions, we can expect to achieve a higher error detection rate.

Second, to further reduce the effect of the garbage collector on detecting errors, it would be possible to modify it to use memory defensively to expect memory errors and recover from them. This is very similar to the construction of the memory scrubber task in high-availability operating systems.

Third, it would be interesting to investigate further the relationship between consumption rates and susceptibility. While both factors depend largely on the application workload and its input, we would like to understand further any correlations or classifications of susceptibility to consumption rates.

7.1 Handling Memory Errors With Java

Java provides an elegant exception programming model through the use of `try/catch` blocks [1]. One future path for investigation would be to consider supporting this exception mechanism to signal memory errors to applications interested in providing error recovery or application state tidy-up on exit. Such support may be of great interest to fault tolerant Java applications, Java databases and Java persistent systems.

When a memory error occurs it can either affect the JVM's or the application's integrity. Determining whether the error affected the JVM or the Java application is fairly complex because the JVM's state is stored both inside and outside of the heap. We propose that it would be possible that when errors occur in the JVM's data areas outside the heap, the JVM could throw an asynchronous `UnrecoverableMemoryError` exception. This is similar to the existing `VirtualMachineError` exception. This could allow for cleaner fail-over handling between redundant machines.

Errors in the VM's heap structures are much more serious and difficult to detect. While the sensitive memory is small, errors can seriously affect both the VM and application. To achieve a suitable level of coverage all heap structures would need to be fully checksummed and updated on modifications. However, a similar `UnrecoverableMemoryError` exception could be raised with sufficient detection support.

The majority of memory errors are likely to occur in the state of an object. We propose that in these circumstances it may be possible to raise a `MemoryErrorException`. However, a large question with this approach is limiting the scope for handling which the exception has the execution. The depreciated `Thread.stop()` method highlights some of the concerns. Raising a `MemoryErrorException` should not allow the system to leave the state of objects in an undefined state. Nor should it generate an exception the target cannot be prepared to handle.

We believe one possible solutions to overcome these problems would be to dispatch `MemoryErrorExceptions` to all dependent threads to allow for informed and safer clean-up from the exception. Since this is an internal VM exception, all threads should be prepared to handle it if they so desire.

However, handling such an exception mechanism is probably too complex to use throughout an application. So it is proposed that to limit the scope to where it is most useful, application programmers could wrap only critical code with such exception handling. Critical sections such as outgoing RPC/RMI access or database accesses would make good candidates since they may hold reproducable transactions and could benefit in improved reliability from this approach. Exceptions occuring at other times can resort to using such exceptions for application clean-up to improve graceful exit/ restart when state is lost.

Clearly support for this exception handling is very complex and poses interesting challenges in performance, coverage, and support. We would like to see research undertaken to investigate this aspect further.

8 Summary

In this paper, we have described our work in studying the memory error susceptibility of the Kaffe virtual machine using fault injection. We found that for the Kaffe VM and the benchmark applications we ran, that heap objects comprise most of the memory error consumption. We also presented our prototype implementation for detecting silent data corruptions by object checksum. We found that this simple technique can detect up to nearly 40% of all application errors caused by silent data corruption.

All experiments were executed in Kaffe's interpretive mode. In order to use Kaffe with its superior performance JIT compiler, the JIT would need to be modified to generate the checksum routine inline with object accesses. Given that errors can occur in any memory, it would also be possible to consider checksumming the generated code, if its size proves this to be necessary. Apart from this, Kaffe using its JIT should have the same overall behavior as has been has reported here, because the same heap management system is used.

While introducing extra overhead of between 32-57% might seem counter to today's JIT research, this overhead represents an upper-bound on performance loss. On the IA-64 architecture, performance can be improved by perhaps four times, because of the ability to use multiple arithmetic units explicitly to parallelize the computation compared to IA-32 architecture processors.

Acknowledgments

We are indebted to Peter Markstein and Ira Greenberg for commenting on the context and presentation of the paper. Together their help significantly improved the document.

References

[1] Arnold, K., Gosling, J., Holmes, D., "The Java Programming Language," Third Edition, Sun Microsystems, 1999.

[2] Bartlett, J., "A Nonstop Kernel," In *Proc. of the Eighth Symposium on Operating Systems Principles, Asilomar, Ca*, pp. 22-29, Dec. 1981.

[3] Brown, N. S. and Pradhan, D.K. "Processor- and Memory- Based Checkpoint And Rollback Recovery," *IEEE Computer*, pp. 22-31, Feb. 1993.

[4] Chen, P. M., et al., "The Rio File Cache: Surviving Operating System Crashes," *Proc. of the 7th ASPLOS*, pp. 74-83, Oct. 1996.

[5] Compaq Corp., "Product description for Tandem Nonstop Kernel 3.0.", http://www.tandem.com/prod_des/tdnsk3pd/tdnsk3pd.htm, Feb. 2000.

[6] Dell, T. J., "A White Paper on the benefits of Chipkill - Correct ECC for PC Server Main Memory," *IBM Microelectronics Division*, Nov. 1997.

[7] Gray, J., and Reuter, A., "Transaction Processing: Concepts and Techniques," Morgan Kaufmann, 1993.

[8] Intel Corp., "Intel IA-64 Architecture Software Developer's Manual," Volume 2, 1999.

[9] Intel Corp., "Intel IA-32 Architecture Software Developer's Manual," Volume 3, 1999.

[10] Kao, W., et al., "Fine: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults," *IEEE T-SE*, vol. 19, no.11, Nov. 1993.

[11] Kermarrec, A-M., et al., "A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability," In *Proc. of the 25th FTCS*, pp. 289-298, June 1995.

[12] Milojicic, D., et al., "Increasing Relevance of memory Hardware Errors – A Case for Recoverable Programming Models," In *Proc of the 9th ACM SIGOPS European Workshop*, Sep. 2000.

[13] Murphy, B., et al. "Windows 2000 Dependability," In *Proc. of the IEEE Intl Conference on Dependable Systems and Networks*, Jun. 2000.

[14] Murphy, B., et al. "Measuring System and Software Reliability using an Automated Data Collection Process," *Quality and Reliability Engineering Intl.*, vol 11, pp 341-353, 1995.

[15] Nick, J.M., et al., "S/390 Cluster Technology: Parallel Sysplex," *IBM Systems Journal*, vol. 36, no 2., pp 172-201, 1997.

[16] Pfister, G., "In Search of Clusters," Prentice Hall, 1998.

[17] Satyanarayanan, et al. "Lightweight Recoverable Virtual Memory," In *Proc. of the SOSP*, pp 146-160, Dec. 1993.

[18] Standard Performance Evaluation Corp. (SPEC) "SPECjvm98 Specification," Aug. 1998. http://www.spec.org/osg/jvm98/

[19] Tandem, Compaq Corporation, "Data Integrity for Compaq NonStop Himalaya Servers," White Paper, 1999.

[20] Tweedie, S. "Designing a Linux Cluster," *Technical White Paper*, Red Hat, Jan. 2000. Also see: http://www.linux-ha.org/

[21] Ziegler, J. F "IBM experiments in soft fails in computer electronics 1978-1994," *IBM Journal of R & D*, vol. 40, no. 1, pp. 1-136, Jan. 1996.

[22] Ziegler, J. F. "Terrestrial cosmic rays," IBM Journal of Research and Development, vol. 40, no. 1, pp. 19-40, Jan. 1996.

Implementing Fast Java™ Monitors with Relaxed-Locks

David Dice
Sun Microsystems, Inc.
Burlington, MA
dice@computer.org

Abstract

The Java™ Programming Language permits synchronization operations (lock, unlock, wait, notify) on any object. Synchronization is very common in applications and is endemic in the library code upon which applications depend. It is therefore critical that a monitor implementation be both space-efficient and time-efficient. We present a locking protocol, the *Relaxed-Lock*, that satisfies those requirements. The Relaxed-Lock is reasonably compact, using only one machine word in the object header. It is fast, requiring in the uncontested case only one atomic compare-and-swap to lock a monitor and no atomic instructions to release a monitor. The Relaxed-Lock protocol is unique in that it admits a benign data race in the monitor unlock path (hence its name) but detects and recovers from the race and thus maintains correct mutual exclusion. We also introduce speculative deflation, a mechanism for releasing a monitor when it is no longer needed.

1. Introduction

The Java language [1] provides monitors [2][12] as the primary synchronization mechanism. Any object can be synchronized upon. In practice, however, most objects pass their lifetimes without ever being involved in synchronization activities. A Java Virtual Machine, or JVM, [3] must therefore provide the *potential* for synchronization but at a very low per-object space overhead. Synchronization operations occur frequently in applications and in the Java runtime library, albeit limited to a small subset of extant objects. It is thus critical that these operations have low latency to avoid degrading performance. Recent research in escape analysis and synchronization removal shows promise, but these techniques apply only to uncontended locking. Even though several recent papers have been published on JVM synchronization, the present work shows that substantial performance improvements can still be made.

The goals for our monitor implementation are thus space-efficiency, time-efficiency and scalability. The design described in this paper scales well to large numbers of threads and processors. It shows low latency for uncontended monitor operations and provides high throughput for contended monitors. It is space-efficient because it uses only one header word per object, and that header word may contain the object's hashCode value as well.

The Relaxed-Lock protocol is used to implement Java-level monitors. It is private to the JVM and is not visible to Java programs.

The design described in this paper was implemented in the Sun Laboratories Virtual Machine for Research (ResearchVM). ResearchVM was previously known as EVM, ExactVM and the Java 2 SDK Production Release. [4] describes another monitor implementation, called the Meta-Lock, based on ResearchVM. Our implementation compares favorably to the Meta-Lock, requiring only one atomic instruction and one memory barrier to lock and unlock an uncontended monitor, while the Meta-Lock algorithm requires two atomic instructions. Like the Meta-Lock, the Relaxed-Lock requires a word in the object header for synchronization. Both algorithms overload this header word to contain both the object's hashCode value and the synchronization word.

The Relaxed-Lock protocol has been implemented in ResearchVM for the Solaris™ Operating Environment, SPARC™ Platform Edition and Solaris IA32™. We present details of the SPARC implementation, but the design is portable to other operating systems and processor architectures. The protocol relies on an atomic compare-and-swap instruction (CAS on SPARC or CMPXCHG on IA32 processors).

2. The Basic Locking Protocol

The paper will first present a simplified form of the protocol and then describe optimizations and reduction to practice. This paper focuses on lock and unlock operations (sometimes called enter and exit, respectively)

that provide mutual exclusion, and doesn't discuss the `wait-notify` aspect of Java monitors.

2.1. Overview

Consider a thread attempting to lock an object. In the Relaxed-Lock protocol each object contains a LockWord field, which is either empty or contains a pointer to a monitor record. A Java thread locks an object with an empty LockWord by allocating a monitor record, writing its unique thread identifier value into the monitor record's Owner field and finally inserting the address of the monitor record into the LockWord using CAS. If the CAS succeeds then the thread has locked the object. We call this *inflation*. Inflation associates a monitor record with an object. If the lock attempt fails, the thread must block itself. To unlock an object upon which no other threads are blocked we deflate the object – we store the empty value into the object's LockWord and return the monitor record to the free pool. We deflate to conserve monitor records. Deflation breaks the relationship between a monitor record and an object.

Now consider multiple threads locking and unlocking a single object where those operations overlap in time. A thread locks an object that is already inflated by applying CAS to try to insert its unique identifier into the monitor record's Owner field. The locking thread may observe that the object's LockWord field refers to a particular monitor record *m*. Since the locking thread fetched this value, the LockWord may have changed – the unlocking thread may have deflated the object and returned *m* to the free pool. Other locking threads may have re-inflated the object with a different monitor record in the meantime. The monitor record *m* may have been reused and may now be associated with a different object. The locking thread holds a *stale* monitor record pointer. The Relaxed-Lock protocol tolerates references to the wrong monitor record. The locking thread will detect the stale pointer and recover as needed.

2.2. Data Structures

The primary data structures related to synchronization are the Execution Environment (*EE*), the object header and the monitor record (*monRec*). Unless otherwise noted, each is initially zero-filled. These structures are implementation specific and are not directly addressable by Java programs.

2.2.1. Execution Environment

An EE is a JVM internal data structure associated with each Java thread. Each EE is attached to a Solaris user-level thread. ResearchVM currently uses a 1:1 thread model – each Java-level thread runs on top of its own Solaris thread. There is only one field in the EE related to synchronization – namely, MonFree, a pointer to the EE's private monitor free list. We use the term thread and EE interchangeably. We say an EE is *waiting* when it has called an object's `wait` method and is awaiting notification. We say an EE is *blocked* when the thread is queued (non-ready) while attempting to lock or re-lock a monitor. In ResearchVM, an EE is uniquely identified by its address.

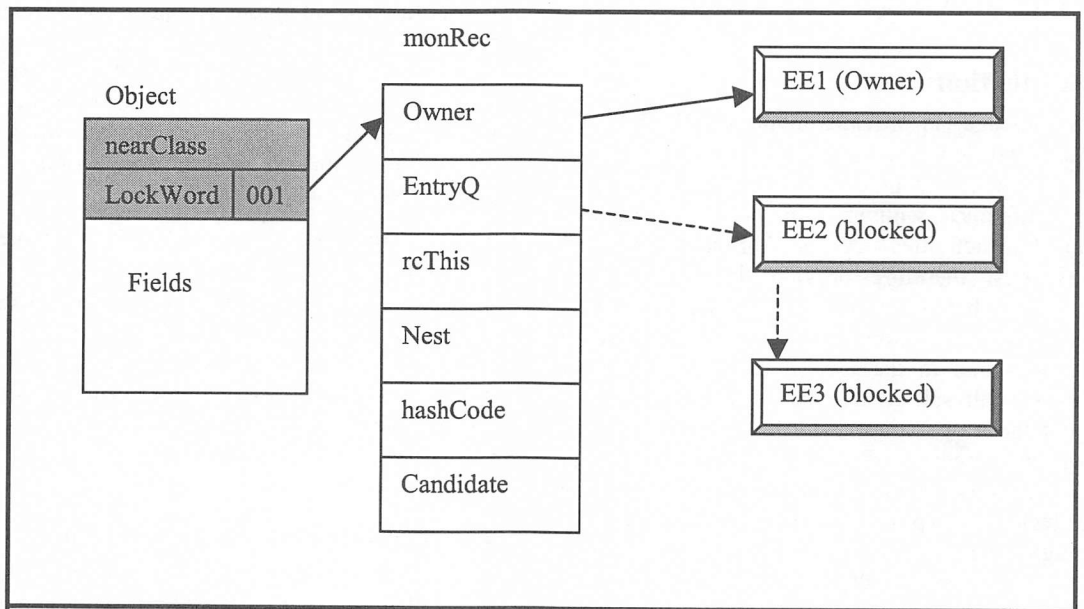


Figure 1. Example of an inflated object. EE1 has locked the object and EE2 and EE2 are blocked on entry.

2.2.2. Object Header

Objects in ResearchVM consist of an object header followed by the constituent fields. The object header consists of a nearClass field, which is used to identify the object type and to implement virtual function calls, and a LockWord field for synchronization. The object constructor initializes the LockWord field to zero. In ResearchVM an object reference is simply a pointer to the object's header.

We distinguish the LockWord state by the low-order bits of the LockWord field. All monRecs are allocated on 8-byte aligned addresses. As such the low-order 3 bits of monRec addresses are known to be zero, allowing encoding of up to 8 states. We use 2: In the *empty* state the LockWord value is zero. All objects are created in the empty state. In the *inflated* state the LockWord contains a pointer to a monRec.

An object may be associated with at most one monRec. Likewise, a monRec may be associated with at most one object at any one time. Many EEs, however, may block on a single object. An EE may block or wait on at most one object at a time. We say an object is locked when the LockWord points to a monRec having a non-NULL Owner field. We say an object is unlocked when its LockWord is empty or its LockWord points to a monRec having a NULL Owner field. We say EE1 owns object *o* when *o*'s LockWord points to some monRec *m* and *m*'s Owner field is set to EE1. An object must be inflated when it is locked or when any threads block or wait on it. Figure 1 shows an example: EE1 owns the object and EE2 and EE2 are blocked on the object.

2.2.3. Monitor Record

The *monRec* is an optional extension to the object header. If space consumption weren't critical, our design would simply embed all the monRec fields directly in the object header itself. Instead, we extend the object on-demand by storing a pointer to a monRec in the LockWord. The monRec is only needed when the object is locked or when threads are waiting or blocking on the object. Like the EE, the monRec is a JVM internal structure and is not a first-class Java object.

A monRec is either free or in use. It is free if it is not associated with any object, and, conversely, it is in use if it associated with an object. Free monRecs reside on per-EE free lists or on a JVM global free list. When a locking thread needs to allocate a monRec with which to inflate an object, it checks its own free list and then, failing that, resorts to the global free list.

The MonFree list is private to the EE. No synchronization is needed to add or remove elements as the list is only accessed by the EE itself. Under normal circumstances, the maximum number of monRecs that an EE needs on its free list is the maximum number of objects that it will lock simultaneously plus one on which to wait. Most EEs reach steady state early in their tenure and never need to grow their lists. In the presence of contention, however, monRecs can migrate between EE free lists. This can result in some EEs "hoarding" monRecs and having an undue number of monRecs in circulation. Unchecked, this could result in unbounded monRec allocation. To compensate, we can either trim and rebalance the free lists at garbage collection (GC) time or, alternatively, mark each monRec with the EE that created it. If an EE deflates an object and observes that the monRec was created by some other EE, it could return that monRec to either the global free list or a special per-EE lookaside free list. The per-EE list is strictly an optimization, but an important one. The Relaxed-Lock protocol would still work correctly with only the global free list.

The major fields in the monRec structure are as follows:

- Owner
  The Owner field is a pointer to the EE that has locked the monRec. The field is NULL if no thread owns the monRec – that is, if the object is unlocked.
- Candidate
  The Candidate field is used to implement *futile wakeup throttling*. See below.
- rcThis
  rcThis is a reference count which indicates the number of threads blocked or waiting on the monRec. The rcThis value does not include the current lock owner itself, if there is one. The rcThis field protects a monRec against inadvertent deflation. The field is updated using atomic fetch-and-add instructions.
- Nest
  Java monitors permit recursive (reentrant) locking. A Java thread can re-lock an object that it already owns without blocking. Nest, initially one after inflation, reflects the recursion depth.
- EntryQ
  The EntryQ is a heavyweight Solaris system semaphore [9] on which we block threads attempting to lock the monRec.

Our implementation requires that the monRecs reside in *type-stable-memory*, or TSM [6][7]. Simply put, once a memory location holds a monRec it must always hold a monRec; the memory is now typed. A monRec must tolerate references from threads that hold stale monRec pointers. The referencing thread will, by examining the monRec and the object LockWord, discover that the reference is stale with respect to the object and recover as needed. We can relax the strict TSM requirement

somewhat in ResearchVM. ResearchVM supports a *stop-the-world* GC model. At a stop-the-world point all Java threads are blocked at known safe points – no thread can be executing in a code path where it holds a stale monRec pointer. At these points we can safely reclaim free monRecs and reuse the underlying memory.

2.3. Run-time support

The following example illustrates the binding between Java source code, Java bytecode and the JVM internal service routines. The Java compiler, javac, translates the Java source code into bytecode. The just-in-time compiler (JIT) in the JVM translates the bytecode into native code at run-time. Execution of monitorenter and monitorexit bytecodes results in calls to `monEnter` and `monExit`, which are "C" routines within our JVM. Java provides both `synchronized` methods and `synchronized` statements. The Relaxed-Lock protocol services both forms in the same manner.

Java Source Code

```
[1]   synchronized (obj) { obj.Value ++ ; }
```

Java Bytecode

```
[1]   push obj
[2]   monitorenter
[3]   push obj
[4]   dup
[5]   getfield #Value
[6]   iconst_1
[7]   iadd
[8]   putfield #Value
[9]   push obj
[10]  monitorexit
```

SPARC Native Code

```
[1]   mov   ee, %o0       ! ee is the current thread
[2]   call  monEnter      ! call monEnter (ee,obj)
[3]   mov   obj, %o1      ! delay slot, pass object
[4]   ld    [obj].Value, tmp
[5]   add   tmp,1,tmp     ! increment obj.Value
[6]   st    tmp, [obj].Value
[7]   mov   ee, %o0       ! ee is the current thread
[8]   call  monExit       ! call monExit (ee,obj)
[9]   mov   obj, %o1      ! delay slot, pass object
```

2.3.1. monEnter

A thread can lock an object in one of two ways:

(1) By Inflation

A thread can lock an object that is deflated by successfully inflating the object. The thread uses CAS to attempt to install a monRec in the object's LockWord. The monRec's Owner field has been set to the thread's EE prior to the CAS A successful CAS confers ownership.

(2) By directly locking the associated monRec.

A thread can lock an object that is already inflated by using CAS to transition the Owner field from NULL to the thread's own EE address.

Under the Relaxed-Lock protocol all internal locking is potentially contended. Consider a thread blocked trying to lock a monitor. After waking, the thread must recontend for a monitor by either inflating the object or by attempting to CAS the Owner field in the associated monRec. Waking up doesn't imply ownership of a lock, but rather grants that thread an opportunity to compete for the lock. Blocking and waking threads is simply a way to avoid spinning.

The Relaxed-Lock protocol uses a two-level synchronization model. Java synchronization primitives that don't involve blocking or waking threads are satisfied in the JVM itself. We call this *fast path* synchronization. Locking an uncontended monitor, for instance, requires executing an atomic CAS instruction in the JVM but doesn't involve any heavyweight system synchronization services. On the SPARC processor the fast path lock primitive is only 13 instructions long. The JVM only resorts to the *slow path*, which uses heavyweight Solaris synchronization primitives [9], to explicitly block or wake threads. Most synchronization requests are satisfied via the fast path.

Locking proceeds as follows. First, the `monEnter` routine fetches the object LockWord and examines the low-order bits, which encode the state.

Case 1: Empty (fast path)

The locking thread must allocate a new monRec to install in the object's LockWord [Listing 1, Line 33]. It checks, in the following order, the EE's monRec free list, the global list and, if still unsatisfied, finally constructs a monRec using `malloc`. A subtle but important point is that all monRec on a thread's free list are known to have their Nest field preset to one and the Owner field preset to the EE. Such monRecs are immediately ready to be installed in the object's LockWord field.

Once a monRec is allocated we try to inflate the object by using CAS to install the monRec's address in the LockWord [Listing 1, Line 37]. If the CAS fails then we've encountered interference; another thread changed the LockWord between the initial load and the CAS. In the case of interference, we simply restart the `monEnter` routine and retry the entire operation. The algorithm is lock-free [10] as at least one thread will have made forward progress. Successfully installing the monRec confers ownership to the calling thread. For uncontested locking this is the most frequently executed

path and it largely determines synchronization performance.

Case 2: Inflated and already locked by the calling thread

If the LockWord is inflated then the LockWord value contains a pointer to a monRec. We examine the Owner field in the monRec. If it is equal to the caller's EE then this is a case of reentrant locking. We simply increment the monRec's Nest field and return. This path requires no atomic operations [Listing 1, Line 46].

Case 3: Inflated and unlocked

As above, we convert the LockWord value to a monRec pointer. In this case the monRec's Owner field is NULL, indicating the object is unlocked. This can occur when there are threads waiting for notification on the object but the object is unlocked. As there are waiting threads, the monRec's rcThis value will be greater than zero. The acquiring thread attempts to transition the Owner field from NULL to its EE via CAS. If the CAS fails then we fall into the slow path (case 4). If the CAS succeeds then we've locked a monRec, although not necessarily the correct monRec.

Between the initial fetch of the LockWord and the successful CAS, there is a timing window during which another thread may have changed the object's LockWord. We check for this by re-fetching the LockWord and verifying that it is still the same [Listing 1, Line 52]. If the LockWord is unchanged then the thread has successfully locked the object – `monEnter` returns. If the LockWord has changed then the thread has locked the wrong monRec. We say that the monRec pointer is stale with respect to the object. In this case the thread releases the monRec, wakes up any threads that may have blocked because they observed that the monitor was locked, and, finally, retries the entire operation. Transiently locking the wrong monRec is harmless.

Case 3 is an optimization and is not strictly necessary to the protocol. We encounter Case 3 when one or more threads are waiting on an object, the object is unlocked and our thread attempts to the lock the object. Case 3 allows us to lock an object with only one atomic instruction. The protocol would still work correctly if we eliminated case 3 and simply used case 4 (the slow path). Case 4 can handle any inflated object but requires a minimum of 3 atomic instructions.

Case 4: inflated and locked by another thread

The object is contended and we now take the slow path and prepare to block the calling thread. First, `monEnter` atomically increments the rcThis reference count field to indicate that another thread is blocked on the monRec. [Listing 1, Line 64]. While incrementing the reference count another thread may have broken the associating between the object and the monRec. We check for this by re-fetching the object's LockWord and insuring it still points to the expected monRec. This is similar to Valois' `SAFEREAD` technique [10]. Next, we resample the Owner field to see if the owner thread relinquished the object between the original fetch of the LockWord and this point. This closes a timing window and prevents lost wakeups. If the Owner field was observed to be NULL the acquiring thread applies CAS to replace the NULL value with its EE address [Listing 1, Line 74]. If the CAS succeeds then the thread has locked the object; we decrement the rcThis field, set the Nest field to one and return. Otherwise, the thread failed to lock the monitor and it now blocks itself on a heavyweight Solaris semaphore associated with the monRec.

Upon waking, the thread checks to see if has been *flushed*. Flushing is an exceptional condition and is described in more detail below, in the `monExit` section. A thread can wake either because it's been flushed or, normally, because the prior owner has released the monitor and arranged that this thread awake as the "heir apparent" owner of the object. If the object's LockWord still points to the expected monRec then the thread attempts to lock the object by using CAS to store its EE into the Owner field. If the CAS succeeds the thread has locked the object – it then decrements the reference count, sets the Nest field to 1 and returns. If the CAS fails, the thread simply re-blocks on the semaphore. If the thread observes that the object's LockWord has changed, then it has been flushed. The pointer to the monRec is stale; the locking thread decrements the rcThis reference count and restarts the entire `monEnter` path.

2.3.2. MonExit

The `monExit` routine releases the monitor associated with an object [Listing 1, Line 86]. In the Relaxed-Lock protocol only the owner of an object may deflate it, and then only at unlock time. `MonExit` starts by fetching the object's LockWord. If the LockWord is uninflated or if the current thread is not the Owner of the object then `monExit` throws an *IllegalMonitorState* exception as required by the Java Language Specification. `monEnter` then decrements the Nest field. If the monitor is recursively locked, `monExit` simply updates the monRec's Nest field and returns [Listing 1, Line 116].

If decrementing the Nest field results in zero then we must release the object. Note that we don't store the zero value in the Nest field. We leave the value at one to avoid the store – the monRec is immediately ready for its next incarnation and can be added to the per-EE free list without any further processing. If the unlocking thread

observes that the rcThis field is non-zero then it will leave the object inflated as there are other threads legitimately waiting on the monitor. The unlocking thread then marks the Owner field as NULL and, to ensure progress, wakes up one of the threads attempting to lock the monitor.

If the unlocking thread observed that the rcThis field was zero then there *appear* to be no other threads referencing the monitor [Listing 1, Line 96]. In this case we attempt *speculative deflation*. First, we deflate the object by restoring the empty value into the LockWord. This dissociates the object and the monRec. Next, after restoring the LockWord, the unlocking thread re-fetches the rcThis field. In the normal case, the value will still be zero and the thread simply adds the monRec to its free list and returns. If, however, the rcThis field is non-zero then we've misspeculated and must take special action.

We call this speculative deflation, as the unlocking thread previously observed that rcThis was zero, but that might not remain true at the actual point of deflation. In the time between fetching rcThis and deflating the object, another thread may have arrived in `monEnter`, attempted entry and incremented rcThis. The entering thread may also have blocked itself on the monRec's EntryQ. Put another way, the code in `monExit` could inadvertently deflate an object while another thread was in the process of trying to lock that same object. The result of misspeculation is that a locking thread could block on a stale monRec with undesirable consequences – the thread could be stranded indefinitely or it could lock the wrong monRec and violate the mutual exclusion constraint. To recover from misspeculation we must flush the monRec:

Flushing in the unlocking thread:

> The unlocking thread detects misspeculation by noticing that the rcThis field changed from zero to non-zero during deflation. This indicates that threads tried to lock the object while it was deflating. The unlocking thread then wakes up all the threads that attempted to lock the object during the timing window. In our implementation, the flushing thread must wait for all the flushees (victims) to acknowledge the flush. This way the monRec's Owner field stays non-zero and tardy threads can't inadvertently lock the wrong monRec. Once the flush is completed and all blocked threads are known to have vacated the monitor the unlocking thread adds the monRec to its private free list.

Flushing in the locking thread:

> The locking thread wakes up and notices that the object's LockWord field is not the expected value. This indicates that the thread has been flushed. When a thread recognizes that it has been flushed, it decrements the monRec's reference count field and retries the entire `monEnter` operation.

Misspeculation occurs rarely as the window of vulnerability is short. [Listing 1, Lines 96-97]. On SPARC, the window – in `monExit`, between the fetch of the rcThis field and the deflating store into the object's LockWord field – is only 6 instructions long. By tolerating this timing window we are able to remove all atomic instructions from the uncontested unlock path. Instead of preventing references to stale monRecs we detect and recover as needed. Misspeculation can't occur when only one thread is accessing an object (no contention) or when the many threads block on an object, in which case the object remains inflated (heavy contention).

The protocol deflates monitors in order to limit the number of monRecs in circulation. An alternative to speculative deflation is to defer deflation until GC-time. The collector could scan and deflate as needed. Deflating at GC-time also makes for a very slightly faster unlock path as we don't need to check reference counts or to deflate. Our implementation uses speculative deflation, however, as it is less coupled to the garbage collection subsystem. In addition, speculative deflation is aggressive, and deflates an object as soon as possible. This minimizes the number of monRec in circulation. Contrast this to Bacon's *Thin Lock* scheme [8][15] where objects, once inflated, stay inflated for their lifetime.

The rcThis field is a hint used to guide deflation. If the rcThis is value is non-zero in `monExit` [Listing 1, Line 96], then the monRec is in use and is not eligible for deflation. It is safe to sample rcThis before releasing the object – while the object is locked rcThis can only increase; it can never transition to zero. We are not at risk of missing deflation and leaking monRecs. If rcThis is zero then the monRec is idle and is eligible for deflation.

2.3.3. Wait-Notify

The `wait-notify-notifyAll` portion of the monitor subsystem isn't described in this paper. We should note, however, that the monRec's rcThis includes the number of threads waiting on the monitor. If there are any threads `waiting` on an object then the object must be inflated. The wait-notify subsystem is largely decoupled from the lock-unlock portion of the synchronization protocol.

3. Augmenting the Basic Protocol

3.1. Safely incrementing rcThis.

As described above the Relaxed-Lock protocol suffers a timing window that permits monRecs to leak. Consider

a `monEnter` call that encounters an inflated and locked monitor. The locking thread must increment the monRec's rcThis reference counter. It may, however, have incremented the rcThis value of a stale monRec pointer. The locking thread recognizes that the pointer is stale and compensates by decrementing the rcThis field. Unfortunately an exiting thread could have observed the rcThis field when it was temporarily (and improperly) non-zero. This, in turn, could cause `monExit` to miss the deflation of the object currently associated with the monRec. Missed deflation would result in leaking monRecs.

To compensate for this timing window `monEnter` puts the potentially leaked monRecs onto a special SuspectList [Listing 1, Line 67]. A monRec is suspect if it may have missed being speculatively deflated. The GC thread scans and cleans the SuspectList at GC-time. Specifically, the GC thread will examine the monRec and the associated object and perform any deflation that may have been missed. The GC thread is able to safely perform the recovery because at GC-time all the Java threads will be stopped at known locations. Because we use a stop-the-world GC mechanism, when the GC thread executes we know that no normal Java threads will be holding stale monRec pointers or executing in a vulnerable region.

The SuspectList allows us to detect and recover from the timing window in monEnter. In the full-length paper we describe another technique that *prevents* a thread holding a stale monRec pointer from incrementing the rcThis field. The idea is based on *word-tearing*; we use mixed-size load, store and CAS operations to access collocated fields in the monRec. Briefly, we define a composite field in the monRec structure. The composite field contains two subfields, rcThis and the Guard, that are separately addressable with load and store instructions. The composite field is addressable with atomic load, store and CAS instructions. The basic idea is the Guard changes when the monRec recycles (deflates). To increment the rcThis subfield we use CAS on the composite field. The CAS will fail if the monRec is stale. The CAS simultaneously validates that the monRec has not recycled and conditionally increments the reference count. Word-tearing is processor and memory-model dependent. It is not portable but is known to work on current SPARC and Pentium™ processors.

3.2. hashCode multiplexing

We now provide a brief sketch of the changes needed to let the LockWord and the object's hashCode cohabit in one header word.

Each Java object may have a hashCode value associated with it [5]. The hashCode, once assigned, is persistent with respect to that object. The hashCode values for a set of objects should have a reasonable distribution as they are often used as keys for hash tables.

ResearchVM realizes hashCode values on-demand. As such, an object's LockWord may be empty, contain the hashCode value or be inflated. We use the low order bits to distinguish the contents. At inflation time, in `monEnter`, the locking thread copies the hashCode from the LockWord and stores it in the monRec [Listing 1, Line 35]. At deflation time the unlocking thread copies the hashCode back into the LockWord [Listing 1, Line 97]. [4] attributes the idea of the displaced header words to Lars Bak in the HotSpot VM. To avoid timing hazards, the first time we inflate an object, if a hashCode has not been assigned, we generate a hashCode and associate it with an object. An inflated object always has a hashCode value.

We compute the hashCode as the XOR of the object's current address and a global gcHash value. This calculation is extremely fast and doesn't impact synchronization performance. The gcHash value is recomputed using a Park-Miller [16] random number generator at each GC-point. In a sense, the address provides a spatial component to the hashCode and the gcHash value contributes a temporal component. The gcHash component is particularly important as, when using a copying garbage collector, heap addresses tend to be reused and don't have a good distribution.

3.3. Futile wakeup throttling

Consider the following policies used to activate a successor thread when a thread unlocks an object. In *directed handoff* the unlocking thread explicitly picks a successor from the list of blocked threads, marks that thread as the owner of the object and then wakes it. The distinguished successor, by virtue of waking, knows that it owns the object. Directed handoff is strictly fair, assuming a LIFO list. When multiple threads repeatedly contend for the same object, however, the directed handoff policy results in high levels of context switching. Assume a typical parallel program that executes the following loop: lock a shared object, execute serial work, unlock the object, execute parallel work. If multiple threads execute the loop they will contend for the shared lock. If the locking is fine-grained and duration of the "execute parallel work" and "execute serial work" phases is short then cost of the context switching will dominate performance

In *competitive handoff*, when a thread unlocks an object, it marks the object as available and then makes a *potential* successor thread (sometimes called the heir apparent) ready. The successor, upon waking up, must

compete for the object like other threads. Waking a successor ensures progress. Competitive handoff is inherently unfair as one thread may dominate the lock. By avoiding excessive context switching and by keeping "hot" threads running it usually provides the best system throughput. Competitive handoff relies on system-level thread preemption to provide a coarse level of fairness. The ResearchVM Meta-Lock and the Relaxed-Lock protocol both use competitive handoff.

Competitive handoff suffers from the futile wakeup problem. When multiple threads repeatedly compete for the same lock, one thread tends to dominate and the remaining threads tend to migrate back and forth between the monitor's EntryQ and the system ready queue. In particular, the successor threads will often wake up, fail to grab the lock and re-block on the monitor. It fails to acquire the lock because the previous owner, which made the successor ready, has reacquired the lock in the interim. This is a *futile wakeup*. The threads eventually make progress, but suffer from degraded performance because of the excessive context switching. To avoid this effect the Relaxed-Lock protocol uses *futile wakeup throttling*. Instead of permitting an unbounded number of successors to be ready, we permit at most one. At any one time, we only need one heir apparent to ensure that the computation makes forward progress. Having more than one heir apparent is unnecessary and inefficient. Throttling greatly reduces the futile wakeup rate.

To implement throttling we use a field in the monRec called Candidate. Candidate is set to one to indicate that the next time the thread releases the monitor it should also wakeup a successor. The field is set to zero to indicate that no wakeup is needed. To be precise, zero means that either no threads are blocked on the monitor or that a successor has been make ready but has not yet come awake. Throttling greatly improves the performance of Java applications that have many threads contending heavily for a single object. [Listing 1, Lines 19 and 72]. Throttling is an optimization and is not fundamental to the Relaxed-Lock scheme.

To demonstrate the utility of throttling consider a Java program, RandBash, that has 24 threads, running in parallel, each of which loops calling the `java.util.Random.nextInt` method for a shared object 1000000 times. The `nextInt` method calls a worker routine that is synchronized. On an 8-way 333 MHz SPARC system running Solaris 2.8 we have the following results:

JVM	seconds
ResearchVM with Meta-Lock	141
ResearchVM with Relaxed-Lock, throttling enabled	40
ResearchVM with Relaxed-Lock, throttling disabled	119

Table 1 Throttling Results.

3.4. Fast Assembly Language Paths

In order to improve performance, our implementation, like Meta-Lock, uses fast, specialized forms of `monEnter` and `monExit` to handle uncontested locking. These routines are written in assembly language. Space permits us from describing these further.

4. Results

	JVM		
Benchmark	Meta-Lock	Relaxed-Lock	
VolanoMark 2.1.2 -- The VolanoMark benchmark, created by Volano LLC, predicts the performance of an internet chat server. We tested it with the rooms parameter set to 10 and the message count parameter set to 100. VolanoMark executes a large number of uncontended synchronization operations.	14925	15346	msgs/sec
pBOB 1.2 (Portable BOB) -- pBOB was created by IBM to model the performance of object databases. It uses a random number generator to create the synthetic workload. The random number generator class is protected by a static monitor and is highly contended. pBOB scores reflect the throughput of a large number of threads passing through a single critical section.	49282	137349	tpmBOB

Contend – Contend, like pBOB, tests the ability of a JVM to handle high levels of contention on a single monitor. It uses 24 threads executing concurrently, each of which iterates 100000 time over a loop. The loop body consists of a parallel portion, which takes 1.55 µsecs to complete and a serial portion, protected by a global monitor, which also requires 1.55 µsecs to complete.		23.771	8.993	secs
SPECjvm98				
_201_compress	LZW compress and decompress	39.205	39.209	secs
_202_jess	Java Expert Systems Shell	18.278	17.316	secs
_209_db	Simulates a database – search and update	60.931	59.045	secs
_213_javac	Java source to bytecode compiler	32.659	31.867	secs
_222_mpegaudio	Compress an audio file	42.916	41.602	secs
_227_mtrt	Multithreaded ray tracer	11.466	10.288	secs
_228_jack	Self-generating parser generator	23.643	22.456	secs
Java Grande Forum Benchmark, Version 2.0. Section1:Method:Same:SynchronizedInstance		5027309	6727437	calls/sec
Sync – Sync is a single-threaded micorbenchmark that times the execution of uncontested synchronization operations. The synchronized statements and methods are empty.				
1M normal calls to an empty method		30	30	msecs
No waiting threads, 1M synchronized method calls		251	179	msecs
No waiting threads, 1M synchronized method calls – nested		125	114	msecs
No waiting threads, 1M synchronized statements		244	207	msecs
One waiting thread, 1M synchronized method calls		451	246	msecs
One waiting thread, 1M synchronized method calls – nested		296	114	msecs
One waiting thread, 1M synchronized statements		462	212	msecs
CHashMapTest – CHashMapTest, written by Doug Lea, exercises his mostly-concurrent reading, exclusive writing HashMap package. As run, it has 4 threads concurrently applying 1000000 updates each to a HashMap having 100000 elements.		19.249	5.733	secs

Table 2 Benchmark Results.

All tests were run on an 8-Way 333MHz SPARC system running Solaris 2.8 in the Interactive Scheduling class. The Relaxed-Lock JVM used fast assembly language paths, speculative deflation, hashCode multiplexing and futile wakeup throttling.

As shown above, the Relaxed-lock protocol takes 179 milliseconds to complete 1 million calls to an empty synchronized method. For comparison, optimized "C" code runs 1 million `mutex_lock-mutex_unlock` pairs in 236 milliseconds. We should note, however, that for "C" code a large component of the cost is the control transfer through the procedure linkage table.

5. Conclusions

5.1. Recap

We have presented the Relaxed-Lock protocol that supports Java monitor semantics. It has been validated in the context of ResearchVM. It is time-efficient, reasonably space-efficient and holds up well under contention. For uncontended locking it requires only one atomic CAS to lock an object and only a memory barrier to unlock and object. On most processors atomic instructions are very expensive so the number of atomic instructions in the lock-unlock code path determines synchronization performance. ResearchVM with Relaxed-Lock actually has a longer lock-unlock path in terms of instruction count than the Meta-Lock form, but the Relaxed form has lower latency and can sustain a higher throughput because it

has one less atomic instruction. As we use fewer atomic instructions, The Relaxed-Lock protocol incurs less memory bus traffic and scales better on multiprocessor systems. It also provides predictable performance and is free of pathologies. The synchronization portion of the JVM stands alone and is largely independent of the rest of the JVM, and in particular the garbage collector.

5.2. Future work

In the future we may investigate using model checkers to formally validate the Relaxed-Lock protocol. Also of interest is making the monRec a first-class Java object. This would greatly simplify management of the monRec pool. In the current implementation the monRec contains a Solaris semaphore, and therefore can't be moved by the garbage collector. To avoid this problem we'd put a system semaphore in each EE and do away with the semaphore in the monRec. An EE would always block on its own semaphore. The monRec EntryQ would then become a pointer to an explicit list of EEs blocked on an object. Meta-Lock uses an explicit linked list of EEs. This would also give the JVM considerable control over short term scheduling policy. We would also be able to eliminate the rcThis field and use the explicit linked list pointer as indication that the monRec was idle.

Flushing, while rare, victimizes the unlocking thread. To keep the monRec out of circulation the exiting thread must wait until all flushees rendezvous. Intuitively, this seems unfair. One remedy would be to hand off the monRec to a special thread dedicated to flushing.

5.3. Acknowledgements

I'd like to thank Ole Agesen, Paula J. Bishop, Alex Garthwaite, Maurice Herlihy, Paul Hohensee and Doug Lea for useful suggestions.

6. References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series, Addison-Wesley, 1996.

[2] C.A.R. Hoare. Monitors: An Operating Systems Structuring Concept. *CACM* 17(10), pp. 549, October 1974.

[3] Timothy Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series, Addison-Wesley, 1996.

[4] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y.S. Ramakrishna, Derek White. An Efficient Meta-Lock for Implementing Ubiquitous Synchronization.. In *Proceedings of ACM SIGPLAN '99 Conference on Object-Oriented Programming Languages, Systems and Applications* (OOPSLA), 1999.

[5] Ole Agesen. Space and Time-Efficient Hashing of Garbage-Collected Objects. *Theory and Practice of Object Systems*, Volume 5, Number 2, 1999, pp. 119.

[6] Michael Greenwald and David Cheriton. The Synergy Between Non-Blocking Synchronization and Operating Systems Structure. *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI)*, USENIX, Seattle, October 1996, pp. 123

[7] Michael Greenwald. Ph. D. Thesis. Non-Blocking Synchronization and System Design. Stanford University, 1999.

[8] David F. Bacon, Ravi Konuru, Chet Murthy and Mauricio Serrano. Thin Locks: Featherweight Synchronization for Java. *In Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998. pp. 258.

[9] B. Lewis, D. Berg. *The Threads Primer: A Guide to Multithreaded Programming*. Sunsoft Press, Prentice Hall, 1996.

[10] John Valois. Lock-Free Data Structures. Ph. D. Thesis, Rensselaer Polytechnic Institute, 1995.

[11] David L. Weaver, Tom Germond, editors. *The SPARC Architecture Manual, Version 9*. SPARC International, Prentice-Hall, 1994.

[12] Peter A. Buhr, Michel Fortier, Michael H. Coffin. Monitor Classification. *ACM Computing Surveys*, 27(1), pp. 63-107. March 1995.

[13] Y. Oyama, K. Taura, A. Yonezawa. Executing Parallel Programs with Synchronization Bottlenecks Efficiently. University of Tokyo, 1998.

[14] Tamiya Onodera, Kiyokuni Kawachiya. A Study of Locking Objects with Bimodal Fields. In *Proceedings of ACM SIGPLAN '99 Conference on Object-Oriented Programming Languages, Systems and Applications* (OOPSLA), 1999.

[15] R. Dimpsey, R. Arora, K. Kuiper. Java server performance: A case study of building efficient, scalable JVMs. IBM Systems Journal, Volume 39, no. 1, 2000 – *Java Performance.*

[16] S.K.Park, K.W. Miller. Random Number Generators: Good Ones Are Hard to Find. *CACM* 31(10), pp. 1192, October 1988.

7. Appendix – Listing

```
[1]   typedef struct _monRec {                  // monitor record
[2]       struct _ExecEnv * volatile Owner ;    // EE, NULL iff free
[3]       volatile int Candidate ;              // wait indicator -- futile wakeup throttle
[4]       volatile int rcThis ;                 // ref count -- holds monRec persistent
[5]       int Nest ;                            // recursive locking
[6]       int hashCode ;                        // hoisted from object.LockWord
[7]       struct _monRec * MonNext ;            // LL chain: next free monRec
[8]       lwp_sema_t EntryQ ;                   // heavy system sync object used for entry
[9]       // Other fields elided …
[10]  } monRec ;
[11]
[12]  typedef struct _ExecEnv {                 // EE
[13]      monRec * volatile MonFree ;           // Linked list of free monitors
[14]      // Other fields elided ...
[15]  } ExecEnv ;
[16]
[17]  void monWakeup (ExecEnv * ee, monRec * m)
[18]  {
[19]      if (m->Candidate == 1 && CAS(&m->Candidate, 1, 0) == 1) {
[20]          _lwp_sema_post (&m->EntryQ) ;
[21]      }
[22]  }
[23]
[24]  int monEnter (ExecEnv * ee, Object * obj)
[25]  {
[26]      monRec * m ;
[27]      monRec * nxt ;
[28]      intptr_t raw;
[29]
[30]    retry:
[31]      raw = obj->LockWord ;
[32]      if (!INFLATED(raw)) {             // Case 1: Empyt - fast path
[33]          m = ee->MonFree ;
[34]          if (m == NULL) m = ExtendFreeList (ee);
[35]          m ->hashCode = raw ;
[36]          nxt = m->MonNext ;
[37]          if (CAS(&obj->LockWord, raw, MKMON(m)) == raw) {
[38]              ee->MonFree = nxt ;       // successfully installing "m" confers ownership
[39]              return OK;
[40]          }
[41]          goto retry ;                  // CAS failed: interference - retry
[42]      }
[43]
[44]      m = MONREC(raw) ;
[45]      if (m->Owner == ee) {             // Case 2: Inflated and locked by calling thread
[46]          m->Nest ++ ;
[47]          return OK ;
[48]      }
[49]
[50]      // Case 3: inflated and unlocked, optimization - not strictly necessary.
[51]      if (m->Owner == NULL && CAS(&m->Owner, NULL, ee) == NULL) {
[52]          if (obj->LockWord == raw)  {
[53]              m->Nest = 1 ;
[54]              return OK ;
[55]          }
[56]          m->Owner = NULL ;         // "m" is stale wrt obj. Recover as needed
[57]          MEMBAR(StoreLoad) ;
[58]          monWakeup (ee, m) ;
[59]          goto retry ;
[60]      }
```

```
[61]
[62]        // Case 4: inflated and locked by another thread.
[63]        // Slow path ... apparent contention: do this the hard way
[64]        Adjust (&m->rcThis, 1) ;          // atomic fetch-and-add
[65]        if (obj->LockWord != raw) {       // Similar to Valois' SAFEREAD
[66]            Adjust (&m->rcThis, -1) ;     // "m" is stale wrt obj
[67]            MarkSuspect (ee, m) ;
[68]            goto retry ;
[69]        }
[70]
[71]        while (obj->LockWord == raw) {
[72]            m->Candidate = 1 ;
[73]            MEMBAR(StoreLoad) ;
[74]            if (m->Owner == NULL && CAS(&m->Owner, NULL, ee) == NULL) {
[75]                Adjust (&m->rcThis, -1) ;
[76]                m->Nest = 1 ;
[77]                return OK ;
[78]            }
[79]            _lwp_sema_wait (&m->EntryQ) ;
[80]            // We're awake - recontend for the object
[81]        }
[82]        FlushAcknowledge (ee, m) ;  // we've been flushed
[83]        goto retry ;
[84]    }
[85]
[86]    int monExit (ExecEnv * ee, Object * obj)
[87]    {
[88]        monRec * m ;
[89]        intptr_t raw ;
[90]        int nn ;
[91]        raw = obj->LockWord ;
[92]        m = MONREC(raw) ;
[93]        if (INFLATED(raw) && m->Owner == ee) {
[94]            nn = m->Nest - 1 ;
[95]            if (nn == 0) {
[96]                if (m->rcThis == 0) {
[97]                    obj->LockWord = m->hashCode ;   // attempt speculative deflate
[98]                    MEMBAR(StoreLoad) ;             // publish store
[99]                    if (m->rcThis == 0) {           // resample ref count
[100]                       ReturnToFreeList (ee, m) ;  // recycle m
[101]                       return OK ;                 // fast path exit
[102]                   } else {
[103]                       FlushAndFree (ee, m) ;      // misspeculated - expel blocked threads
[104]                       return OK ;
[105]                   }
[106]               } else {
[107]                   m->Owner = NULL ;
[108]                   MEMBAR(StoreLoad) ;
[109]                   monWakeup (ee, m) ;
[110]                   return OK ;
[111]               }
[112]           }
[113]           // The following memory barrier is unrelated to the Relaxed-Lock protocol.
[114]           // The Java Memory Model promises release consistency.
[115]           MEMBAR(StoreLoad);
[116]           m->Nest = nn ;
[117]           return OK ;
[118]       }
[119]       return THROW(ILLEGAL_MONITOR_STATE) ;
[120]   }
[121]
```

Listing 1 "C" Code for monEnter() and monExit()

An Executable Formal Java Virtual Machine Thread Model

J Strother Moore
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712
moore@cs.utexas.edu
George M. Porter
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712
george@cs.utexas.edu

Abstract

We discuss an axiomatic description of a simple abstract machine similar to the Java Virtual Machine (JVM). Our model supports classes, with fields and bytecoded methods, and a representative sampling of JVM bytecodes for basic operations for both data and control. The `GETFIELD` and `PUTFIELD` instructions accurately model inheritance, as does the `INVOKEVIRTUAL` instruction. Our model supports multiple threads, synchronized methods, and monitors. Our current model is inadequate or inaccurate in many respects (e.g., we do not formalize the JVM's finite arithmetic nor do we describe class loading and initialization). But the model is a useful tool for studying the application of formal reasoning to the JVM and to Java programs.

We demonstrate two useful aspects of an operational formal semantics. First, the model is executable: bytecoded methods can be run on the model. Second, the model allows us to prove theorems about those methods or, more generally, about the model. Because the JVM provides a relatively clean semantics for Java, our model can be thought of as a step towards Java software verification. We illustrate these points. We cite some theorems proved about our model, including a theorem involving unbounded multi-threading and mutual exclusion with `MONITORENTER` and `MONITOREXIT`. Our proofs are carried out with the ACL2 theorem prover.

Keywords: parallel, distributed computation, mutual exclusion, operational semantics, verification, JVM, bytecode verification

1 Formal Executable Models

"Formal Methods" is the name given to the computer science research area devoted to the use of formal mathematical logic to model and analyze the properties of computing systems. One advantage of modeling a system formally is that proofs about it can be checked by mechanical proof checkers. This increases the odds that the proofs are flawless. Automated mechanical theorem provers can be used to help discover proofs, which can significantly reduce the tedium of constructing formal proofs.

This is not pie-in-the-sky formal methods proposal boilerplate. It is happening. At Advanced Micro Devices, Inc., formal models of the hardware designs for the floating-point FDIV instruction on the AMD Athlon™ have been mechanically proved to be compliant with the IEEE-754 standard. Indeed, all of the elementary floating-point operations on the Athlon (including addition, subtraction, multiplication, division, and square root) have been so proved. Important security properties of the IBM 4758 secure co-processor were mechanically verified at IBM Yorktown. The correctness of an auditor that checks the output of a compiler for

safety-critical trainborn real-time control software for Union Switch and Signal was mechanically proved. A bit- and cycle-accurate model of a Motorola digital signal processor was mechanically proved to conform to a higher-level sequential view in which the pipeline was abstracted away – provided the microcode being executed was free of a well-defined set of hazards. Several microcoded DSP algorithms extracted from the ROM of that microprocessor were mechanically proved correct. These applications, and others, are described in [12]. All were modeled and verified using one theorem proving system, ACL2.

ACL2 [13] stands for "A Computational Logic for Applicative Common Lisp." It is a functional programming language, a first-order mathematical logic, and a mechanical theorem prover. ACL2 was written by Matt Kaufmann and J Strother Moore (an author of this paper) and is the successor of the Boyer-Moore theorem prover Nqthm [3, 5].

As a programming language, ACL2 is a version of Common Lisp. It provides the familiar Lisp data objects, including numbers, strings, symbols and lists, along with if-then-else and function application, including recursion. ACL2 is axiomatically described. For example, it is an axiom that `(IF x y z)` is `z` if `x` is `NIL`, and is `y` otherwise. Another axiom is that `(car (cons x y))` is `x`. Theorems about ACL2 functions can be proved in this logic, most often by case analysis, simplification, and mathematical induction. A mechanical tool has been built to help the human user construct proofs. This interactive computer program combines term rewriting, decision procedures and a wide variety of heuristic techniques to provide a symbolic manipulation system. The system has sophisticated automatic search strategies for finding certain kinds of proofs and those strategies can be informed and guided by advice from the user, most often in the form of key lemmas suggested by the user and proved by the system. For details, see [13] and the ACL2 Web site `http://www.cs.utexas.edu/users/moore/acl2`. In this paper we avoid ACL2 syntax and knowledge of ACL2 insofar as possible.

The techniques for modeling microprocessors and programming languages in such a logic have been developed over a long period of time in the Boyer-Moore community. A tour de force of the method is presented in the so-called CLI Stack (produced by Computational Logic, Inc.) [1, 8, 18] which is a hierarchy of verified components including a microprocessor, loader, linker, assembler, two compilers, an operating system and some applications programs, all quite simple but also actually fabricated and practical, and all of which have been formally specified and mechanically proved correct. Another example is the work of Yuan Yu, in which the Motorola 68020 microprocessor is formalized. Yu's work is sufficiently accurate that it is possible to compile 21 of the 22 programs in the Berkeley C String Library, using `gcc -o`, and run the resulting binaries on the formal model, computing the expected results. Furthermore, Yu formally specified what these 21 programs were supposed to do and used the Boyer-Moore theorem prover to prove mechanically that the binaries met the specifications [6]. For an introduction to the modeling and proof methods used in these projects, see [4]. We merely hint at the techniques as we briefly describe our model of the JVM.

Of particular historical importance to the present work is Rich Cohen's ACL2 model of a single-threaded JVM [7]. The so-called "defensive JVM" is an accurate and complete model of a subset of the JVM instruction set. As such, the machine is more complicated than the one discussed here, but does not support threads. The defensive JVM checks the dynamic conditions required to insure type safety and is an essential step toward the specification and verification of the Java bytecode verifier. Both Cohen's model and ours are based largely on the Sun Microsystems documentation for Java and the JVM [14, 9], informed by private conversations with experts and experience with Java and the JVM.

Also of special interest is the fact that the JEM1 microprocessor, the world's first silicon JVM, built by Rockwell Collins, was modeled formally with ACL2 [19, 11]. Some proofs were done with the model but its primary use was as a simulator. The ACL2 model executes at about 90% of the speed of a carefully-written C simulator for the same model. The issues involved in the efficient execution of ACL2 models are discussed in the article by Greve, Wilding, and Hardin (Chapter 8) of [12].

There is a large body of academic work on

Java modeling but relatively little that is truly formal and still less that is supported by mechanized tools. A wonderful exception is the work by Nelson, Leino and others at Compaq Systems Research Center on the "Extended Static Checker" for Java, which is formal, practical and mechanized. See `http://research.compaq.com/SRC/esc/`. The work of Borger and Schulte [2] on Java exceptions is quite formal and accurate, but not supported by mechanized proofs. Mechanically checked proofs about simple Java programs have been constructed with several theorem provers, including HOL, Isabelle, and PVS. See, for example, [17]. However, we are unaware of mechanically checked proofs (other than those reported here) of Java classes that use multi-threading. Our work is distinguished primarily by being cast in a formally defined operational (and executable) semantics. Because we formalize the semantics we can prove theorems about the model, not just about particular Java methods or classes. We know of no mechanically checked proofs (ours included) of correctness properties of significant Java applications; the field is still in its infancy.

2 Specification of the JVM

In ACL2, machines are formalized by adopting an explicit representation of the states and then writing an interpreter for the machine language. Another way of putting it is this: to formalize a machine language, implement a simulator for it in functional Lisp. While this may seem to be a mere programming exercise, it is also a logic exercise if the simulator is written in an axiomatically described programming language like ACL2.

In our model of the JVM, a state consists of three components: the thread table, the heap, and the class table. We describe each in turn. When we use the word "table" here we generally mean a list of pairs in which "keys" (which might be thought of as constituting the left-hand column of the table) are paired with "values" (the right-hand column of the table). Such a table is a map from the keys to the corresponding values.

The thread table maps thread numbers to threads. Each thread consists of three components: a call stack, a flag indicating whether the thread is scheduled, and the heap address of an object of class `Thread` in the heap uniquely associated with this thread. We discuss the heap below.

The call stack is a list of frames treated as a stack (the first element of the list is the topmost frame). Each frame contains five components: a program counter and the bytecoded method body, a table associating variable names with values, a stack, and a synchronization flag indicating whether the method currently executing is synchronized. Unlike the JVM, the local variables of a method are referenced by symbolic names rather than positions.

The heap is a table associating heap addresses with instance objects. An instance object is a table. The keys of an instance object are the successive classes in the superclass chain of the object. The value of each such key is another table, mapping the immediate field names of the class to their values. The structure of heap addresses is unimportant but they can be distinguished from integers and other data types. In our model a heap address is a list of the form `(REF` i`)`, where i is a natural number. One point where our model differs from the JVM is that in our model the `NEW` instruction is completely responsible for the object's instantiation; all fields are initialized to `0`. Classes in our model do not have separate constructors.

Finally, the class table is a table mapping class names to class descriptions. A class description contains a list of its superclass names, a list of its immediate fields, and a list of its methods. We do not model syntactic typing in our machine, though we could. Thus, our list of fields is just a simple list of field names (strings) rather than, say, a table mapping field names to signatures. A method is a list containing a method name, the names of the formal parameters of the method, a synchronization status flag, and a list of bytecoded instructions. Our model omits signatures and the access modes of methods.

Bytecoded instructions are represented abstractly as lists consisting of a symbolic opcode name followed by zero or more operands. For example, `(LOAD X)` is the instruction that pushes the value of local variable `X` onto the stack in the current frame. `(ADD)` pops two items off

the stack in the current frame and pushes their sum. `(IFEQ 12)` pops an item off the stack and if it is `0`, increments the program counter by `12`; otherwise it increments it by `1`. The similarity of these instructions to certain JVM instructions should be obvious, as should be the differences: we ignore the different types of LOAD (e.g., ILOAD, DLOAD, etc.) and ADD instructions, we ignore the finite range of integer data, and we count program counter offsets in number of instructions rather than number of bytes. These and most of the other discrepancies between the current model and the JVM are matters of detail that would not change the basic structure of the model to fix and do not impact our ability to use the model to study proof techniques.

For those readers curious to see how we define the semantics of such operations in ACL2, see Table 1. It contains the definition of the function `execute-PUSH` which we use to give semantics to the PUSH instruction. The instruction `(PUSH 3)` is comparable to `ICONST_3` or `BIPUSH 3` on the JVM.

The function takes three arguments, named `inst`, `s`, and `th`. The first is the list expression denoting the instruction. The first element of `inst` will always be the symbol PUSH and the second is the constant that is to be pushed on the stack of the current frame. The second argument of `execute-PUSH`, `s`, is the JVM state, consisting of a thread table, a heap and a class table. The third argument, `th`, is the number of the thread that is to be "stepped." `Execute-PUSH` returns the state obtained by executing the PUSH instruction in the given thread of `s`. It creates that state with the function `make-state`, which takes three arguments: the thread table, the heap and the class table of the state to be returned. The last two components of the new state above are the same as those in `s`. The thread table is modified by replacing the entry for `th` by another entry. That entry's call stack is obtained by replacing the topmost frame of the current call stack (notice we push a frame onto a stack obtained by popping one off). In the new frame, the program counter is advanced by 1, the locals remain unchanged, the constant (extracted from `inst` using the function `arg1`) is pushed on the stack, and the method program and synchronization flag are unchanged.

The most complicated instruction formalized in our model is INVOKEVIRTUAL. An example INVOKEVIRTUAL instruction on our machine is represented by the list structure `(INVOKEVIRTUAL "ColoredPoint" "move" 2)`. Note that in place of the JVM's signature we provide only the number of parameters, since we consistently ignore type issues in this model. We paraphrase the definition of `execute-INVOKEVIRTUAL` by describing the state it creates from an instruction of the form below, a state s, and a thread number th.

`(INVOKEVIRTUAL` c *name* n`)`: Let *ref* be the item n deep in the stack. This is expected to be a heap reference to an instance object, *obj*. Let *class* be the class of this object (the first key in the table, i.e., the name of the most specific class in the object's class hierarchy). Use the function `lookup-method` to determine from the class-table of s the closest method with name *name* in *class* or its superclass chain. Let *formals* and *body* be the formal parameters and bytecoded body of the closest method. Let *formals'* be *formals* with the new symbol `THIS` added to the front.

Create a new call stack, cs', from the call stack of thread th in s by replacing the topmost frame by a new frame in which the program counter has been incremented by one and $n + 1$ items have been popped off the stack. Create another call stack, cs'', by pushing a new frame onto cs'. This new frame should have a program counter of 0 and an empty stack. The locals of the new frame should bind *formals'* to the topmost $n + 1$ items removed from the stack in s (above), the deepest of which is bound to `THIS`. The bytecoded body of the frame should be *body*. We will use cs' and cs'' in various cases below and we will not be interested in cs'' unless the closest method is non-native. Consider the following cases.

- The closest method is native: We support only two native methods, `"start"` and `"stop"` from the `"Object"` class. We describe only the first here. In this case, *obj* should include the class `"Thread"` in its superclass chain. The new state constructed by the `"start"` method has the same heap and class table as s. The thread table is changed in two ways. First, the call stack of th is replaced by cs' above (stepping over the INVOKEVIRTUAL). Second, the thread

```
(defun execute-PUSH (inst s th)
  (make-state
   (modify-tt th
              (push (make-frame (+ 1 (pc (top-frame s th)))
                                (locals (top-frame s th))
                                (push (arg1 inst)
                                      (stack (top-frame s th)))
                                (program (top-frame s th))
                                (sync-flg (top-frame s th)))
                    (pop (call-stack s th)))
              'SCHEDULED
              (thread-table s))
   (heap s)
   (class-table s)))
```

Table 1: execute-PUSH

th′ uniquely associated with *obj* is changed so that its scheduled flag is SCHEDULED.

- The closest method is a synchronized method: Fetch the contents of the "monitor" and "mcount" fields in the "Object" class of *obj*. If the mcount is 0 or the mcount is non-0 but the monitor is *th*, then we say *obj* is "available" to *th*. If *obj* is available to *th*, then the new state is obtained from *s* by replacing the call stack with *cs″* after setting the sync-flg component of the top frame to LOCKED, and by replacing the heap of *s* with a heap in which the "mcount" field of the object at *ref* has been incremented by one and the "monitor" field has been set to *th*. If, on the other hand, *obj* is unavailable, then the "new" state is *s* itself. Thus, the thread hangs at the INVOKEVIRTUAL instruction until *obj* becomes available. We do not specify the scheduler; instead, our model allows all possible interleavings of thread executions and some thread states (as the one just described) make no change if stepped before progress is possible.

- Otherwise, the new state is obtained from *s* by replacing the call stack with *cs″* after setting the sync-flg component of the top frame to UNLOCKED.

Given execute-PUSH, the reader can presumably imagine how this description is coded in ACL2.

We formalize a variety of instructions in this style, including POP, LOAD, STORE, ADD, MUL, GOTO, IFEQ, IFGT, RETURN, XRETURN, NEW, GETFIELD, PUTFIELD, MONITORENTER, and MONITOREXIT. For each such opcode *op* we define an ACL2 function execute-*op* that takes the instruction, current state, and thread number and returns the next state.

We then define step to be the function that takes a state and a thread number and executes the next instruction in the given thread, provided that thread exists and is SCHEDULED. Step is essentially a "big switch" on the opcode of the instruction indicated by the program counter and method body in the top frame of the call stack of the given thread.

Finally we define run to take a "schedule" and a state and return the result of stepping the state according to the given schedule. A schedule is just a list of numbers, indicating which thread is to be stepped next. That is, our model puts no constraints on the JVM thread scheduler; however stepping a non-existent, UNSCHEDULED, or otherwise blocked thread is a no-op. We find it convenient also to define (runn n schedule s) to run the first n steps of schedule starting in state s.

The complete ACL2 source text for our machine is available from http://www.cs.utexas.edu/users/moore/publications/m4/index.html.

Our model omits many features of the JVM.

Among the more glaring omissions are accurate support for the JVM primitive data types like ints, doubles, arrays, etc., support for syntactic typing both in the naming convention in the instruction set (e.g., IADD versus DADD) and field and method signatures, class loading and initialization, INVOKESTATIC (with the concomitant requirement that classes have representative instance objects in the heap upon which synchronization can be arranged), exception handling, and errors. Experience with other commercial microprocessor models leads us to believe that these features could be added to our model without fundamentally changing its basic structure. There is no doubt that they greatly complicate the model and would complicate proofs about programs that use the features in question. That is one of the reasons we left them out. Our model is adequate however as a vehicle for studying basic mechanized proof techniques for dealing with Java programs, including multi-threaded applications.

3 Some Examples of Execution

Because our model, run, is an ACL2 program, it can be executed on concrete data to produce concrete results. To run our bytecode we create a state, say s_0, containing the thread table, heap, and class table we have in mind. An expression constructing such a state is shown in Table 2. The class Alpha contains a single instance method fact which is just a recursive factorial program written in our bytecode.[1] The bytecode is similar to that produced by compiling

```
public int fact(int n) {
 if (n<=0) return 1;
   else return n*fact(n-1);
}
```

except our arithmetic is not bounded. We show the JVM bytecode for this fact in Table 2, in line-by-line correspondence with ours. The main program of the only thread in the thread table of s_0 creates a new instance object of class Alpha and invokes its fact method after pushing 5. That is, it calls fact on 5.

[1] We have not modeled INVOKESTATIC in this machine, so we have chosen to make fact an instance method.

Call the state in Table 2 s_0. To run s_0 we must provide a schedule. Since the main program is in thread 0 (the only thread) and creates no other threads, a suitable schedule is just a list of 0's.

The list constant shown in Table 3 is the state created by evaluating (run (repeat 58 0) s_0). Call that state s_{58}. Thus, s_{58} is the result of stepping thread 0 fifty-eight times starting with s_0. The code elided away is just the definition of our fact method. Inspection of thread 0 reveals that the program counter of the top frame is pointing to the last instruction, the (XRETURN), and that 120 is on the top of the stack. Stepping once would pop the top frame and push the 120 onto the empty stack of the frame below. The new top frame is poised to execute the (HALT) instruction. So stepping s_0 sixty times halts the machine with 120 on top of the stack in the main frame. Since 120 is 5!, the fact method seems to have worked.

Also evident in Table 3 is the representation of the instance object at heap address 0, an object of class Alpha (which has no fields) with superclass "Object" (which has fields "monitor", "mcount" and "wait-set"). Note also the class table, which, in addition to our Alpha class, contains two built in classes, Object and Thread. In our model, the Object class has only the three fields listed, and the Thread class has only two (native) methods, "start" and "stop" whose semantics are built into INVOKEVIRTUAL.

There are of course many schedules that run thread 0 in s_0 for sixty instructions. Any schedule containing sixty 0's would work, no matter how many other thread numbers are interspersed between them.

Because our model is expressed in a formal mathematical logic, it is possible to reason about it formally, using the ACL2 mechanical theorem prover. Rather than just test that the fact method works for 5 and a few other numbers, we can prove a theorem stating that the fact method computes the factorial of its argument.

```
(make-state
 (make-tt                                  ; Thread table
  (push                                    ;  call stack of thread 0
   (make-frame 0                           ;   frame:  pc
               nil                         ;           locals
               nil                         ;           stack
               '((NEW "Alpha")             ;           method body
                 (STORE OBJ)
                 (LOAD OBJ)
                 (PUSH 5)
                 (INVOKEVIRTUAL "Alpha" "fact" 1)
                 (HALT))
               'UNLOCKED)                                sync status
   nil))
 nil                                       ; Heap
 (make-class-def                           ; Class Table
  (list
   (make-class-decl
    "Alpha"                                ;  class name
    '("Object")                            ;  superclasses
    NIL                                    ;  fields
    '(("fact" (N) NIL                      ;  Method int fact(int)
       (LOAD N)                            ;   0 iload_1
       (IFGT 3)                            ;   1 ifgt 6
       (PUSH 1)                            ;   4 iconst_1
       (XRETURN)                           ;   5 ireturn
       (LOAD N)                            ;   6 iload_1
       (LOAD THIS)                         ;   7 aload_0
       (LOAD N)                            ;   8 iload_1
       (PUSH 1)                            ;   9 iconst_1
       (SUB)                               ;  10 isub
       (INVOKEVIRTUAL "Alpha" "fact" 1)    ;  11 invokevirtual #8 <Method int fact(int)>
       (MUL)                               ;  14 imul
       (XRETURN))))))                      ;  15 ireturn
```

Table 2: A State for Computing Factorial

```
(((0                              ; Thread 0
   ((11                           ;  call stack, top frame: pc
     ((THIS . (REF 0)) (N . 5))   ;                           locals
     (120)                        ;                           stack (120 on top)
     ((LOAD N)                    ;                           method body
      (IFGT 3)
      (PUSH 1)
      (XRETURN)
      (LOAD N)
      (LOAD THIS)
      (LOAD N)
      (PUSH 1)
      (SUB)
      (INVOKEVIRTUAL "Alpha" "fact" 1)
      (MUL)
      (XRETURN))                  ;                            (pc points here)
     UNLOCKED)                    ;                           sync status
    (5                            ;  next frame:              pc
     ((OBJ . (REF 0)))            ;                           locals
     ()                           ;                           stack (empty)
     ((NEW "Alpha")               ;                           method body
      (STORE OBJ)
      (LOAD OBJ)
      (PUSH 5)
      (INVOKEVIRTUAL "Alpha" "fact" 1)
      (HALT))                     ;                            (pc points here)
     UNLOCKED))                   ;                           sync status
  SCHEDULED NIL))
((0                               ; Heap address 0:
   ("Alpha")                      ;  Alpha fields:  none
   ("Object"                      ;  Object fields:
    ("monitor" . 0)
    ("mcount" . 0)
    ("wait-set" . 0))))
                                  ; Class table
(("Object"                        ;  Object class
   ()                             ;   superclasses
   ("monitor"                     ;   fields
    "mcount"
    "wait-set")
   (("start" () NIL NIL)          ;   methods
    ("stop"  () NIL NIL)))
 ("Thread"                        ;  Thread class
   ("Object")                     ;   superclasses
   ()                             ;   fields (none)
   (("run" () NIL (RETURN))))     ;   methods
 ("Alpha"                         ;  Alpha class
   ("Object")                     ;   superclasses
   ()                             ;   fields (none)
   (("fact" (N) NIL (LOAD N) ... (XRETURN))))))  ; methods
```

Table 3: The Result of Stepping the Factorial State 58 Times

Theorem. fact *is correct.*

```
(implies (and (poised-to-invoke-fact s th n)
              (natp n))
         (equal (top
                 (stack
                  (top-frame
                   (run (fact-sched n th) s)
                   th)))
                (factorial n)))
```

The hypotheses of the theorem assume that `s` is a state poised (in thread `th`) to invoke our `fact` method on the natural number `n`. Because `fact` is an instance method, this requires inspecting the top two objects on the stack to make sure that the topmost is a natural number and that the resolution of the name `"fact"` in the class of the next item is our `fact` method. The conclusion is an equality stating that a certain expression is equal to `(factorial n)`. Here, `factorial` is the mathematical function of that name, defined in ACL2. The expression in question describes the top item on the stack in the top frame of thread `th` in the state obtained by executing state `s` a certain number of steps, as given by `(fact-sched n th)`. Thus, this theorem establishes that by running thread `th` a certain number of steps, it computes the factorial function.[2]

The proof of the factorial theorem can be constructed interactively with the ACL2 theorem prover and the theorem prover is entirely responsible for the correctness of the proof. In this case, the user provides an inductive argument and the machine carries out that argument, expanding definitions, applying axioms and basic theorems about the machine. For a discussion of such theorems see [15]. The proof that `fact` computes factorial takes about 30 seconds (on a 700 MHz machine).

In Table 5 we show a more interesting state, modeled after the Java `Apprentice` code shown in Table 4. In this state the main program creates an object of class `Container` and then loops forever creating and starting `Thread` objects of class `Job`. Each `Job` is in an infinite loop using the method `incr` to read, increment, and write into the `counter` field of the `Container` object. The critical section of the `incr` method is protected by `MONITORENTER` and `MONITOREXIT`.[3]

[2]Our machine has unbounded integer arithmetic. We could, of course, model Java's bounded arithmetic. The factorial theorem would have to be restated to reflect that.

[3]Our byte code for `"run"` exploits the fact that `"incr"` returns `"this"` and is slightly different than the compiled Java.

If we remove the `MONITORENTER` and `MONITOREXIT` (and the corresponding `LOAD`) instructions from the bytecode (i.e., remove the synchronization from the Java method) we can exhibit a schedule that makes the counter decrease: run the main thread until it has started two jobs, then run the first thread until it pushes the value of the counter (which at this point will be 0) onto its local stack, then run the other thread many cycles to increment the counter several times, and finally run the first job again so that it increments its 0 and writes a 1 into the counter field.

The ability to deal with schedules and states abstractly makes it easier to explore such issues. This illustrates the value of an executable abstract model.

However, there is no schedule that makes the state in Table 5 decrease the counter. This cannot be demonstrated by testing. It can, however, be proved by analyzing our model. Here is a theorem proved with the ACL2 theorem prover about the state shown in Table 5, here called `*a0*`.

Theorem. Apprentice *Monotonicity*

```
(implies (and (natp n)
              (natp m)
              (<= n m))
         (<= (counter
                 (runn n any-schedule *a0*))
             (counter
                 (runn m any-schedule *a0*))))
```

The theorem compares the values of the counter in two states, one obtained by running `*a0*` `n` steps and the other obtained by running `m` steps, both according to the same completely unconstrained schedule. If $n \leq m$, the counter in the former state is less than or equal to that in the latter state. This theorem is a statement about an unbounded number of parallel threads using the JVM synchronization primitives. The proof requires careful (and rather global) analysis of what is happening in the heap. (For example, all threads writing to the `Container` respect the monitor and no thread changes the `objref` field of a running thread.) See, for example, Praxis 56 in [10], where Haggar writes "Do not reassign the object reference of a locked object." For details of our proof see [16].

```
class Container {
    public int counter;
}
class Job extends Thread {
    Container objref;
    Object x;
    public Job incr () {
        synchronized(objref) {
            objref.counter = objref.counter + 1;
        }
        return this;
    }
    public void setref(Container o) {
        objref = o;
    }
    public void run() {
        for (;;) {
            incr();
        }
    }
}
class Apprentice {
    public static void main(String[] args){
        Container container = new Container();
        for (;;) {
           Job job = new Job();
           job.setref(container);
           job.start();
        }
    }
}
```

Table 4: The Apprentice Class: Unbounded Parallelism

```
(make-state
 (make-tt                          ; Thread Table
  (push (make-frame 0              ;  call stack, top frame: pc
                    '((CONTAINER . NIL) (JOB . NIL))    ; locals (uninitialized)
                    ()                                  ; stack (empty)
                    '((NEW "Container")                 ; main method
                      (STORE CONTAINER)
                      (NEW "Job")
                      (STORE JOB)
                      (LOAD JOB)
                      (LOAD CONTAINER)
                      (INVOKEVIRTUAL "Job" "setref" 1)
                      (LOAD JOB)
                      (INVOKEVIRTUAL "Job" "start" 0)
                      (GOTO -7))
                    'UNLOCKED)
        nil))
 nil                               ; Heap
 (make-class-def                   ; Class Table
  (list (make-class-decl "Container"        ;   Container class
                         '("Object")        ;    superclasses
                         '("counter")       ;    fields
                         '())               ;    methods (none)
        (make-class-decl "Job"              ;   Job class
                         '("Thread" "Object");    superclasses
                         '("x" "objref")    ;    fields
                         '(("incr" () nil   ;    methods
                            (LOAD THIS)
                            (GETFIELD "Job" "objref")
                            (STORE TEMP)
                            (LOAD TEMP)
                            (MONITORENTER)
                            (LOAD THIS)
                            (GETFIELD "Job" "objref")
                            (LOAD THIS)
                            (GETFIELD "Job" "objref")
                            (GETFIELD "Container" "counter")
                            (PUSH 1)
                            (ADD)
                            (PUTFIELD "Container" "counter")
                            (LOAD TEMP)
                            (MONITOREXIT)
                            (LOAD THIS)
                            (XRETURN))
                           ("run" (N) nil
                            (LOAD THIS)
                            (INVOKEVIRTUAL "Job" "incr" 0)
                            (GOTO -1))
                           ("setref" (R) nil
                            (LOAD THIS)
                            (LOAD R)
                            (PUTFIELD "Job" "objref")
                            (RETURN))
                           )))))
```

Table 5: Apprentice in Our Model

4 Conclusion

This paper is a first step at developing an executable abstract formal model of threading in the JVM. We have explained how such a model can be built, we have shown that the model can be executed on concrete data to test the behavior of methods and threads under various scheduling regimes, and we have illustrated that it is possible to prove theorems about all possible behaviors. These proofs can be checked mechanically by ACL2, a general-purpose theorem proving engine. This engine was not designed with JVM proofs in mind; indeed, all the engine "knows" about the model is its formal definition. Our proofs of the theorems cited were straightforward applications of general techniques understood well by the ACL2 community. Further investigation of such theorems would undoubtedly lead to the codification of JVM-specific proof techniques and formal metatheorems, such as that syntactically non-interfering threads can be analyzed separately.

The prover is sufficiently powerful to be of use in verifying microprocessor architectures and floating-point implementations. While our JVM model is currently quite simple compared to the implemented JVM, past experience with ACL2 supports the hope that ACL2 will be capable of handling significantly more realistic models of the JVM.

If our past experience is any guide, the formalization of proof techniques for a realistic model of the JVM will make it easier to reason formally about the JVM and Java. In addition, such an undertaking will most likely expose oversights or ambiguities in existing informal understandings of how to write correct and reliable Java programs.

5 Acknowledgments

Our machine, which we call M4 because it is the fourth machine in a series that approaches the JVM in complexity, owes much of its basic structure to Rich Cohen who used ACL2 to formalize a single-threaded version of the "defensive JVM" [7]. We are grateful to Rich for his pioneering effort into the JVM formalization, as well as to the entire ACL2 and Boyer-Moore community for their development of techniques to formalize and reason about such machines. We are also grateful to David Hardin and Pete Manolios, who have each made many valuable suggestions in the course of this work.

6 Availability

The ACL2 system is freely available from the following website:

`http://www.cs.utexas.edu/users/-moore/acl2`

The ACL2 source code for our machine is available from:

`http://www.cs.utexas.edu/users/-moore/publications/m4/index.html`

The authors can be contacted at the e-mail addresses specified on the first page.

References

[1] W.R. Bevier, W.A. Hunt, J S. Moore, and W.D. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989.

[2] E. Borger and W. Schulte. Defining the java virtual machine as platform for provably correct java compilation. In *Proceedings of MFCS'98*, volume LNCS 1450, pages 17–35. Springer, 1998.

[3] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[4] R. S. Boyer and J S. Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Es-*

says in Honor of Larry Wos, pages 147–176. MIT Press, 1996.

[5] R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition*. Academic Press, New York, 1997.

[6] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.

[7] R. M. Cohen. The defensive Java Virtual Machine specification, version 0.53. Technical report, Electronic Data Systems Corp, Austin Technical Services Center, 98 San Jacinto Blvd, Suite 500, Austin, TX 78701, 1997.

[8] A. D. Flatau. A verified implementation of an applicative language with dynamic storage allocation. Phd thesis, University of Texas at Austin, 1992.

[9] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[10] P. Haggar. *Practical Java Programming Language Guide*. Addison-Wesley, 2000.

[11] David Hardin, Matthew Wilding, and David Greve. Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification – CAV '98*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998. See URL `http://pobox.com/users/-hokie/docs/concept.ps`.

[12] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.

[13] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition)*. Addison-Wesley, 1999.

[15] J S. Moore. Proving theorems about Java-like byte code. In E.-R. Olderog and B. Steffen, editors, *Correct System Design – Recent Insights and Advances*, pages 139–162. LNCS 1710, 1999.

[16] J S. Moore and G. Porter. Proving properties of java threads. *(submitted for publication)*, 2000.

[17] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential java programs. In *14th International Workshop on Algebraic Development Techniques (WADT'99)*, volume LNCS 1827, page (to appear). Springer, 2000.

[18] Matthew Wilding. A mechanically verified application for a mechanically verified environment. In Costas Courcoubetis, editor, *Computer-Aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993. See URL ftp://ftp.cs.utexas.edu/pub/boyer/-nqthm/wilding-cav93.ps.

[19] Matthew Wilding, David Greve, and David Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, to appear. Draft TR available as `http://pobox.com/users/-hokie/docs/efm.ps`.

TRaDe, A Topological Approach to On-the-fly Race Detection in Java Programs

Mark Christiaens
ELIS, Ghent University
Gent, 9000, Belgium
mchristi@elis.rug.ac.be

Koen De Bosschere
ELIS, Ghent University
Gent, 9000, Belgium
kdb@elis.rug.ac.be

Abstract

Debugging multi-threaded programs is notoriously hard. Probably the worst type of bug occurring in multi-threaded programs is a data race. There is therefore a great need for tools to automatically detect data races during execution. This article presents, TRaDe, a novel approach to detect races in object-oriented languages using a topological approach. An implementation of TRaDe based on the Sun JVM 1.2.1 is compared with existing tools. TRaDe proves to be a factor 1.6 faster than any known race detection tool for Java and has memory requirements similar to the best competing tools.

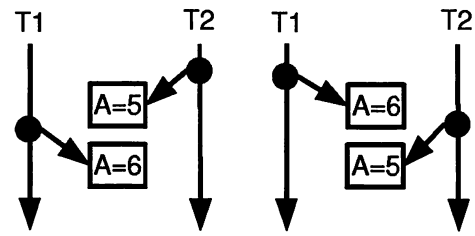


Figure 1: On the left we see thread 2 accessing a common object, A, and writing the value 5. This is followed by thread 1, writing the value 6 to A. The result of this operation is that A contains the value 6. On the right, thread 1 for some reason executes faster which results in the same events happening but in reverse order. This is possible since there is no synchronisation between thread 1 and 2. A now contains the value 5.

1 Introduction

Multi-threaded applications are hard to debug. This is due to the fact that when searching bugs in multi-threaded applications we have to reason about a multitude of threads, each simultaneously performing separate tasks. One particularly hard bug to detect is a 'data race'. A data race occurs when the programmer does not correctly synchronize the access to a variable which is being manipulated by more than one thread. This can leave the variable in an unexpected or inconsistent state. In Figure 1, we see a simple example.

Data races are very hard to find because of two reasons. First of all, they are non-deterministic because they depend on the interleaving of the actions of threads, which is not always the same. Even if we observe them in one run, during a next run, they may not occur again, leaving us totally in the dark as to what went wrong. Secondly, they are non-local. One thread may be performing a spelling check and another may be editing the text being checked. These are two almost totally unrelated sections of code that, if not well synchronised, will cause havoc.

There are three major approaches to finding data races:

- Static analysis of parallel programs has been proven in general to be an NP hard problem [13].
- Post-mortem analysis usually involves large traces of the execution of the multi-threaded program [4, 14, 15] but recently new techniques were developed that make this approach viable [16, 17].
- On-the-fly analysis [5] has no need for traces since it tries to detect data races as they occur. However, it has the potential to be very intrusive which must be avoided as much as possible.

Current on-the-fly techniques incur large overheads due to the fact that they must observe every read and write operation to shared variables. Time overheads as high as a factor 30 are not uncommon.

In this article, we present, TRaDe (Topological RAce Detection), a novel method to automatically detect data races on-the-fly with reduced effort in "pure" object-oriented environments. Using this technique, we are able to dynamically make a selection of the objects we need to observe to find data races by analyzing the graph formed by the interconnection between these objects. This approach is applicable to a wide range of object-oriented languages. Since Java is widely used, object-oriented and multi-threaded, we will give a practical implementation as proof of concept by extending a Java Virtual Machine. We have compared TRaDe to two commercial competitors, JProbe [9] and AssureJ [10]. We have found that AssureJ ignores a subset of data races which we correctly detect. More importantly, TRaDe is on average a factor 1.6 faster than its closest competitors with comparable memory requirements.

In Section 2, we briefly describe the synchronisation primitives of Java and in this context we give our definition of data races. In Section 3, we present the idea of topological race detection followed by a description of our implementation in Section 4. Performance measurements are provided in Section 5. Finally, we indicate some avenues for future research in Section 6 and present our conclusions in Section 7.

2 Synchronisation in Java

Java [2] is an object-oriented language that was designed with multi-threading in mind [11, 12, 21]. In Java there are only two fundamental data types: 'primitive types' and 'reference types'. Primitive types consist of booleans, integers, floats, ... Reference types contain a reference to an object or contain 'null'. These objects are created dynamically on the heap. A garbage collector is responsible for destroying them when they are no longer referenced [8]. Objects themselves contain primitive types or references.

A race between two (or more) threads occurs when they modify a member variable of an object in an unpredictable order. Races on variables on the stack are impossible since the stack can only be manipulated by the thread it belongs to.

To avoid data races, a programmer can force fragments of code running on different threads to execute in a certain order by adding extra synchronisation operations. Java offers several constructs that enforce extra synchronisation[1]:

- `start` and `join` which operate on `Thread` objects,
- locked objects,
- `synchronized` (static) member functions and
- `wait` and `notify(All)`.

The fragments of code of a thread that are separated from each other by a synchronisation operation are abstracted into the notion of 'events'. Notice that the synchronisation operations are not considered events themselves. The i^{th} event of thread T_t will be denoted by $e_{t,i}$. Two events, e_i and e_j, are said to be 'ordered', $e_i \rightarrow e_j$, if there exists a set of synchronisations that force event e_i always to occur before event e_j. A data race occurs when there is no set of synchronisations that force the events modifying a shared variable to occur in a fixed order.

TRaDe models the ordering of events by using a construct called a 'vector clock' as defined in [6, 7, 20]. Vector clocks are tuples of integers with a dimension equal to the maximum degree of parallelism (number of threads) in the application. The first event, $e_{t,0}$ of every thread T_t is assigned the vector clock, $VC(e_{t,0})$, with components

$$VC(e_{t,0})_j = \begin{cases} 0, & j \neq t \\ 1, & j = t \end{cases} \quad (1)$$

The value of the vector clock of the next event in a thread is calculated using the vector clocks of its preceding events. If event $e_{t,i}$ on thread T_t is guaranteed to occur after events $E = \{e_0 \ldots e_n\}$, its vector clock is updated as follows

$$VC(e_{t,i})_j = \begin{cases} max(\{VC(e)_j | e : E\} \cup \{0\}), & j \neq t \\ max(\{VC(e)_j | e : E\} \cup \{0\}) + 1, & j = t \end{cases} \quad (2)$$

[1] There are a few other operations on `Thread` objects, which influence the execution of other threads but which we do not take into consideration since they are either being removed from the Java APIs or cannot be used to synchronize two threads: `destroy`, `interrupt`, `resume` and `stop`.

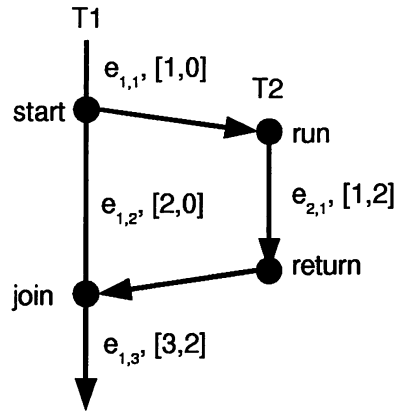


Figure 2: Synchronisation using the `Thread` class

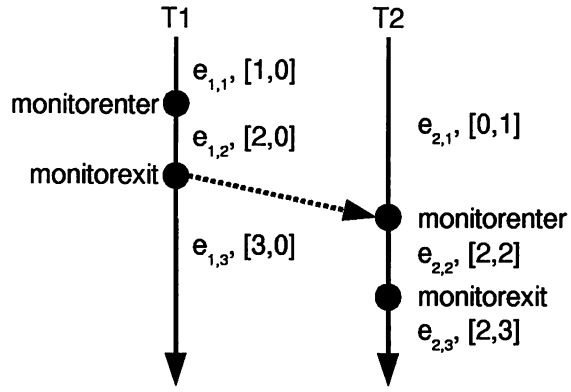


Figure 3: Synchronisation through a locked object

For our purposes, the most important property of vector clocks, is that they can be used to verify whether two events are ordered. Two events, a and b, are ordered iff

$$a \rightarrow b \equiv (\forall i. VC(a)_i \leq VC(b)_i) \wedge a \neq b \quad (3)$$

Two events are parallel, i.e. not ordered, iff

$$a \parallel b \equiv \neg(a \rightarrow b) \wedge \neg(b \rightarrow a) \quad (4)$$

If we define $W(a)$ the set of all locations written to during event a and $R(a)$ the set of all locations read during event a, then two events, a and b, will be involved in a data race iff

$$(a \parallel b) \wedge \begin{array}{l} (W(a) \cap R(b) \neq \emptyset) \vee \\ (R(a) \cap W(b) \neq \emptyset) \vee \\ (W(a) \cap W(b) \neq \emptyset) \end{array} \quad (5)$$

i.e. the two events are executed in parallel, both events access a common variable and at least one event modifies the variable.

The ordering between events is obtained by observing the synchronisation operations in Java. The `start` member function of `Thread` is used by one thread to start the execution of a second thread. The `join` member function allows one thread to wait for the end of the execution of a second thread. These operations impose an ordering on the events of these threads as can be seen in Figure 2. The value of the vector clocks is shown in the figure, illustrating the calculation of the vector clocks at synchronisation operations.

A lock is associated with every `Object` in Java. A thread can try to take this lock using the bytecodes `monitorenter` and `monitorexit`. When the object is already locked, the thread will wait until it is unlocked and can then proceed. This construct does not impose a fixed time order on the code of the two threads involved, it just enforces mutual exclusion. It does suggest that the programmer is aware of a potential race and has used this construct as a means of synchronisation. We therefore consider this a 'de facto' ordering, depicted in Figure 3 by a dashed arrow.

The `synchronized` keyword is applied to a subset of the member functions of a class, the 'monitor'. When a thread invokes one of these member functions on an object of the synchronised class, Java ensures that none of the other member functions in the monitor is being executed. This is implemented through the object locking mechanism mentioned above. When a synchronised member function is executed, the lock of the object containing the member function is taken. When the member function finishes, the lock is released.

A final set of synchronisation primitives is `wait` and `notify(All)` which are member functions of every `Object`. When a thread invokes `wait` on an object, the execution of the thread is halted until another thread executes `notify(All)` on that very same object. At that time the first thread in line can continue its execution. This imposes the ordering depicted by the dotted arrow in Figure 4. However, a thread is only allowed to invoke `wait` or `notify` on an object if that thread owns the lock of that object, so in reality it suffices to observe the ordering between the `monitorenter` and `monitorexit` depicted by the solid arrows.

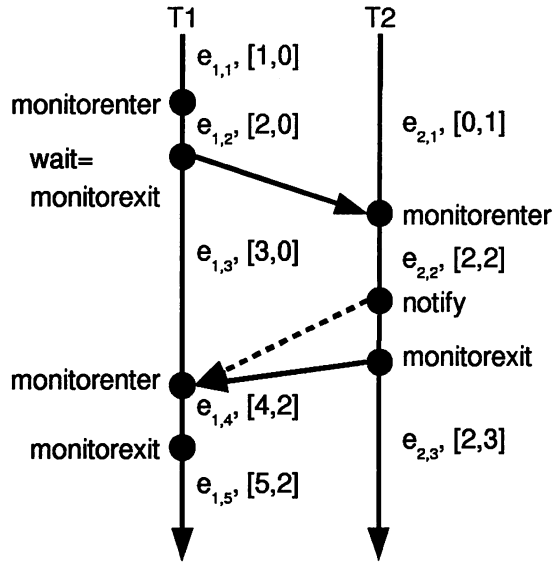


Figure 4: Synchronisation through signals

To detect data races, an access history for every object is constructed. In this access history, every read an write operation to the object must be stored together with the identity of the thread that performed the operation and the vector clock of the event to which the operation belongs. When a new read or write operation occurs, it is compared to the previous operations stored in the access history. If condition 5 is satisfied, a race is found.

Of course, as the program continues to execute, the access histories grow without bounds. In [5] it is shown that it suffices to store only the last[2] write operation in the access history, since these write operations must be ordered or a race would already have occurred. As a consequence, only one write operation is stored in the access history. Also, only the read operations which are parallel with each other need to be stored. So at most one read operation for every thread present in the program must be stored in the access history.

Still, this can amount to a very large overhead, especially if there are many threads (remember that the size of each vector clock grows proportionally to the number of threads). One way out is to reduce the number of objects for which an access history must be maintained. In the next section, we shall present a method which makes this possible.

[2]w.r.t. the vector clock ordering

3 Topological Race Detection

There are some apparent and also some less apparent advantages to doing race detection on the Java bytecodes instead of on the underlying hardware instructions.

First of all, we have the granularity argument. Since Java bytecodes are quite high level, many machine instructions can be necessary to perform one bytecode instruction. If we try to detect data races at the machine level, we will have to observe every hardware read/write instruction for every thread. If however we observe at the bytecode level, many machine instructions can safely be ignored, assuming a correct JVM.

Secondly, Java, by construction, makes a large number of instructions data race free. Every thread has its own private stack on which it allocates local variables and parameters for each member function call. This data can only be modified by the owning thread. As a consequence, all instructions that solely manipulate stack data cannot cause a data race. There are 181 guaranteed data race free instructions and 20 'dangerous' instructions. The latter set can be split into 2 categories. The first category consist of the `{a,b,c,d,f,i,l,s}aload` and `{a,b,c,d,f,i,l,s}astore` bytecodes. These instructions read and write the contents of arrays on the heap. Since objects on the heap can be reached from multiple threads, these instructions need to be checked for data races. The second category consists of `getfield`, `getstatic`, `putfield` and `putstatic`. These are instructions that read or write the fields of objects or classes on the heap.

Finally, and this will prove essential to our technique, Java enforces a very strict object model, even on the bytecode level. Machine instructions can modify practically any location in the address space of a program. Java, on the other hand, only allows modification of an object's data through the reference to that object.

As can be seen in Figure 5, an object is created by invoking the bytecodes `new`, `newarray`, `anewarray` or `multianewarray` (1 in the figure). These instructions all put a reference to the newly allocated object on the stack of the invoking thread. At this point, these objects are only reachable through this one reference and this reference is only accessible to one thread. We call this situation a 'local object'.

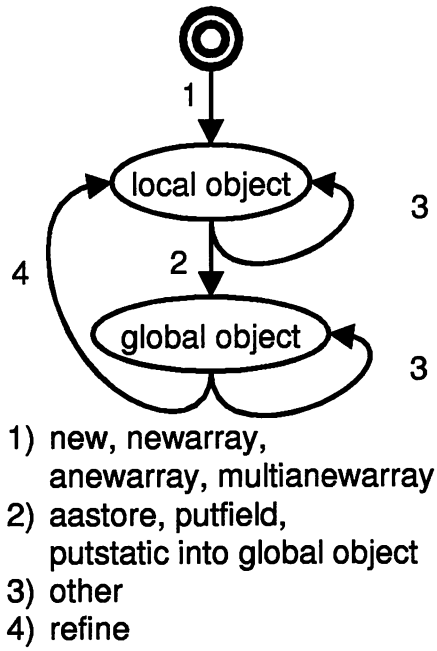


Figure 5: The life cycle of an object

No races are possible. An object that is reachable, through some path, by several threads, is called a 'global object'. It has the potential to be involved in a race. Since we construct information about the reachability of objects from several threads in order to detect potential race problems, we call this method 'topological race detection', TRaDe for short.

Initially an object is created locally. The only references to it exist on its creating thread's stack (1 in Figure 5). One way to change the status of an object from local to global is by storing its reference into a second object. If this second object is reachable by another thread, our object also becomes reachable by this other thread (2 in Figure 5). At this point, the object could potentially be involved in a race. When an object becomes global all the objects reachable from this new global object also become global. If, on the other hand, the second object is solely reachable by our own thread, it remains local.

There are only a small number of bytecodes that can change the topology of the object interconnection graph: `aastore`, `putfield` and `putstatic`. They all have in common that they store references in an object's field.

There are a few exceptions to the outline given above. Every object of type `Class` is global right from the start. The reason for this is that every thread needs to be able to access every class to construct objects of this class. Inside a class, there are static variables that can be read and written to. So these are, by definition of the Java language, immediately global to all threads.

A second way in which an object can become global is when it is involved in the startup of a new thread. In Java, threads are started by creating an object containing a `run` method. When this object's `start` method is called, a new thread is created and starts executing the code in the `run` method. At this very point, this object is reachable by both the new thread and the original thread that started the new thread. It must therefore immediately be made global together with all objects reachable from this object.

This last observation is crucial and is the reason why a refinement of our global objects is necessary (5 in Figure 5). Consider for example the Java program in Figure 6. It creates 10 separate threads (besides the `main` thread). Each of these threads creates a linked list of 10000 local objects.[3] No races are present in the program. Note that the main thread does not maintain a link to the started threads (`g = null`).

This program was artificially made to be very suitable for our approach to race detection; it is very simple and contains large data structures which are clearly not shared between the threads. Almost all calculations are performed on these local data structures so fairly little overhead should be incurred by doing race detection. Still, without the refinement step, TRaDe would perform very badly.

When the new threads are started (at line 29) by invoking `start` on the object `g` of the class `Separate`, `g` must be made global. If it were not for the refinement, these objects would remain global for the remainder of the program's execution. When the threads would start to construct their linked lists, these linked list would also become global. Not much can be gained from such an approach. To do the refinement step, we turned to the garbage collector (GC).

Whatever the underlying algorithm of the GC is, somehow it must determine whether an object is no

[3] Of course the JVM makes hundreds more objects for its internal use but this is not important in this example.

```
1   class Separate extends Thread {
2     Link root;
3     int count;
4
5     class Link { Link prev, next; }
6
7     Separate (int count) {
8       root = new Link ();
9       root.prev = root.next = root;
10      this.count = count;
11    }
12
13    public void run () {
14      for (int i = 0; i < count; i++) {
15        Link l = new Link ();
16
17        l.prev = root;
18        l.next = root.next;
19        root.next.prev = l;
20        root.next = l;
21      }
22    }
23
24    public static void main (String [] arg) {
25      Separate g;
26
27      for (int i = 0; i < 40; i++) {
28        g = new Separate (10000);
29        g.start ();
30      }
31
32      g = null;
33    }
34  }
```

Figure 6: The test program

longer reachable by any thread in the program. If this is the case, the object can be removed from the heap. This is very similar to what TRaDe is trying to do. TRaDe tries to determine whether an object is reachable by *more* than one thread in the program.

We exploited this observation as follows. Each time the GC performs its job, it is followed by our 'refiner'. For every thread, we generate a set, S_i, of all the objects that are reachable from that thread. Then we combine these sets into a set of objects reachable from multiple threads as follows.

$$S_{tot} = \cup_{i,j}(S_i \cap S_j) \qquad (6)$$

We use S_{tot} to refine the general TRaDe mechanism after garbage collection. If an object is not present in S_{tot}, it is only reachable from one thread and therefore local. The large data structures that are necessary to enable data race detection are removed and the object is marked as being reachable only by this one thread.

A minor, yet crucial, modification to our refiner proved necessary. A reference to each `Thread` (and derived classes) is always present in at least one `ThreadGroup` and every `Thread` can obtain the `ThreadGroup` of which it is a member. This has the annoying consequence that every `Thread` can be reached from every other `Thread` through its `ThreadGroup`. If we instruct our refiner to only collect the objects reachable from one thread, it will inevitably start collecting a large portion of the other thread's data structures through these hidden references. The solution was not to follow links leaving `ThreadGroup`. Should a program use these references in `ThreadGroup`, it will circumvent our technique. We believe this will occur only rarely. If this is a problem, we could adapt the JVM further so as to flag such behavior.

Figure 7 compares the approach without a refinement phase (top graph) with the approach with a refinement phase (bottom graph). The figures are constructed by analyzing the heap each time a garbage collection has occurred. We see the total number of handles allocated (a handle points to an object). These handles are subdivided into unused handles which are preallocated each time the heap is expanded, local handles which point to objects that are detected as being local to a thread and global handles which point to objects which were detected as being reachable by more than one thread.

The top graph shows that practically all used handles point to global objects. This means that we would have to observe accesses to most objects for potential data races. The number of local objects is so small that they are not noticeable on the graph.

The bottom graph, using the refiner, shows the expected result. We see that almost all used handles point to local objects. The number of global objects is very small. Clearly, the large linked lists are being detected as local so we will not need to observe these to detect data races.

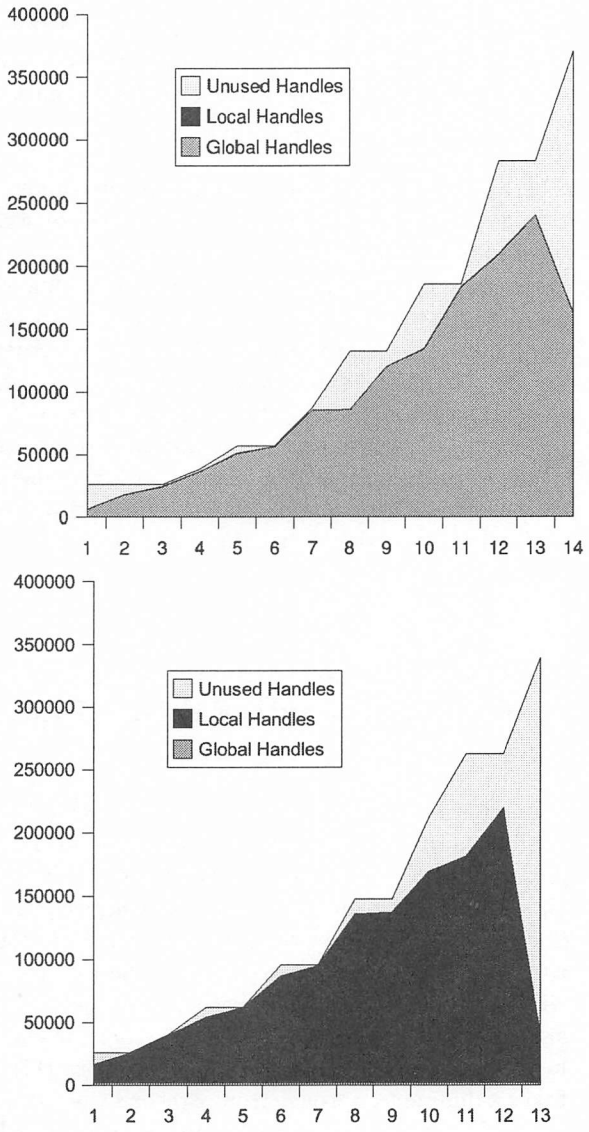


Figure 7: Total number of handles in use measured after each garbage collection phase. The top graph was measured without using the refiner. The bottom graph was measured while using the refiner.

4 Implementation

To test our ideas in practice, we implemented the TRaDe method in an existing JVM. We selected the Sun JVM 1.2.1 on Solaris. This JVM comes equipped with a just in time compiler (JIT). A JIT is used to compile bytecodes to machine instructions on-the-fly so as to accelerate the execution of a Java program. We decided to turn the JIT compiler off to simplify our coding. This way, the JVM just uses an interpreter loop, executing the bytecodes one by one. Our techniques should be readily transferable to JIT compilers.

Our first step was to instrument all the synchronisation primitives of Java using vector clocks as described in Section 2. Vector clocks have a serious drawback: they contain as many components as there are threads in the program. This means that if we are dealing for example with an FTP-server which creates a thread for every file request it receives, the size of the vector clock grows without bounds. We have implemented an advanced version of vector clocks that can dynamically grow *and* shrink as threads are created and destroyed without losing accuracy while doing data race detection and with little overhead involved called 'accordion clocks'. The exact approach we took is beyond the scope of this article.

The next step was to instrument every object with a minimal data structure that allows TRaDe to be used. The basic idea can be seen in Figure 8. When objects are created using `new`, `newarray`, `anewarray` or `multianewarray`, they are extended with a data structure consisting of 20 extra bytes.[4] It consists of 2 parts. The first is the thread identification number (`TID`). In this field, the `TID` of the thread that created this object is stored or, when the object becomes global and is reachable by several threads, -1 is stored. The second part consists of link fields that will be used to link a much larger data structure for full data race detection *only* if the object becomes global.

An object can contain several fields that can be written or read (`#fields`). If we instrument a new global object, each field must have its specific data structure that maintains its access history. This data structure contains: a description (`description`) of the field being accessed (its name,

[4]This could be reduced to 8 bytes, but this is just a prototype.

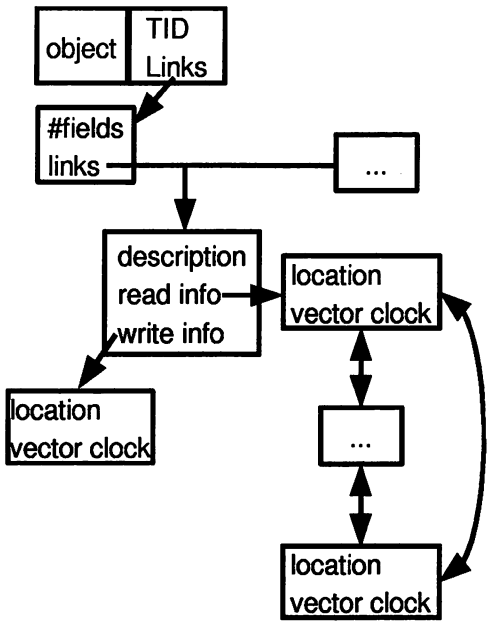


Figure 8: Full instrumentation of an object for data race detection

type info, ...) and links to information about the read and write operations that involved this field. For each read and write operation we store the location in the code where the operation occurred (a class, a member function and a JVM program counter) and a vector clock indicating 'when' the operation occurred.

Using this data structure, the instructions `aastore`, `putfield` and `putstatic` are instrumented. We will explain what happens for `aastore`. A similar procedure is followed for `putfield` and `putstatic`.

Suppose the bytecode `aastore` stores a reference, `R`, into an array, referred to by reference `A`. There are 2 possibilities:

- If the object pointed to by `R` is already global (`R.TID == -1`) then nothing happens, the object is already being watched for possible data races.
- If on the other hand, the object is not yet global, the `TID` of the array referred to by `A` is checked. If it is global (`A.TID == -1`), then by storing `R` into `A`, the object referred to by `R` also becomes global. Otherwise, if `A.TID != R.TID`, we are storing our reference into an array that is reachable by another thread. The object referred to by `R` must again be made global.

If the object referred to by `R` becomes global, we recursively check all its children. Each child that is not yet global is made global. Here we must pay attention to stack overflow when recursively marking a deep data structure as global.

The actual race detection consists of instrumentation added to the 20 bytecodes which read or write to an object as follows. Each time such a bytecode is executed, we check whether it affects a global object. If not, we don't have to do anything; races are impossible. If we are dealing with a global object, we can access the extra data structures and verify using the vector clocks whether this new instruction represents a data race. If so, we flag this to the user. Then we update the access history with the new location of this instruction and the new vector clock indicating when the instruction occurred.

5 Performance Measurements and Comparison with Existing Tools

A number of general tools have already been developed to automatically detect data races in a program. Eraser[5], for example verifies the locking discipline [19]. If a memory location is read/written by different threads, a set of locks must be held. Each time this location is accessed again, the tool checks which locks are held and whether their use is consistent with previous use. Another tool for checking, among others, for data races is RecPlay [17]. It takes a different approach from Eraser's since it performs data race detection off-line, using a recorded trace from a previous execution. The types of races detected are similar to our definition given above.

Both these tools function 'blindly', not knowing what type of program they are analyzing, and observe the stream of processor instructions that are being executed and the memory locations upon which they operate. In contrast, true Java specific tools also exist.

JProbe is a tool capable to detect, among other things, data races [9] in Java programs. It seems

[5]Eraser is apparently now marketed by Compaq under the name VisualThreads

to be using Sun's Java Virtual Machine Profiler Interface (JVMPI). This is an interface that allows profilers to request to be notified of certain events in a Java program such as the loading of classes, start of garbage collection, entering of monitors, etc. Although nothing is published about its internal workings, except its user manual, it seems to use a similar definition of data races as the one used by TRaDe.

Another Java tool is AssureJ [10]. It is also capable of data race detection, among other things, and is very fast. Nothing is known about the algorithm it uses. It does not seem to use the JVMPI but rather to be a modified Sun 1.2 JVM. Again, it seems to use a similar definition of a race as used in TRaDe. One important short-coming that was noticed is that when two events race (so their vector clocks are parallel) but their threads do not actually overlap in time, no race is detected.[6] This is probably due to the fact that they remove all information concerning the behaviour of a thread as soon as it terminates, which is incorrect.

We've run extensive benchmarks comparing TRaDe to JProbe and AssureJ. See Table 1 for a description of the benchmark programs and options used. We selected large applications from a wide variety of problem domains.

The options used for each race detection tool are configured so that each has to find races in the whole of the program. Both JProbe and AssureJ have options to focus their race detection on certain sections of the code. These options were turned off. All programs have the ability to detect races on arrays as a whole, called 'collapsing' arrays, or to consider the elements of an array as separate entities which each can be involved in a race. We chose to collapse arrays to conserve memory since this is the recommended setting for JProbe and AssureJ. Each was configured so as to enable their best performance. All other features were turned off.

The results can be seen in Table 2. All benchmarks were run on a Sun Ultra 5 workstation with 512 MB of memory and a 333 MHz UltraSPARC IIi with a 16 KB L1 cache and a 2MB L2 cache. Memory usage and user time were estimated by averaging 5 measurements with no other user programs running. Some benchmarks could not be run in 512 MB of memory. A larger system was then used as an indication of how much more resources are needed to complete the benchmark. In this case, a Sun Ultra-2 was used with 2048 MB of memory, 4X400 Mhz UltraSPARC IIi processors with 16 KB L1 direct mapped cache and 4MB L2 direct mapped cache each. Note that we were unable to use this machine exclusively. Other user programs were running simultaneously.

We also added some baseline measurements for comparison with normal execution without race detection. Figures using the Hotspot 1.0.1, mixed mode, build f were added. This is a state of the art just-in-time (JIT) compiler from Sun which compiles Java code to native code where necessary. Since we are not using the JIT included in the Sun JVM, we also added baseline figures using only the interpreter version of the JVM.

As can be seen, TRaDe is faster on all benchmarks than JProbe and AssureJ. It beats JProbe by a very large margin; many benchmarks cannot be completed due to memory exhaustion. When averaged out, TRaDe is a factor 1.6 faster than AssureJ. As to memory consumption, TRaDe again beats JProbe by a very large margin.

AssureJ is more on par with TRaDe in the area of memory consumption. AssureJ, on average, uses only a factor 0.74 of the memory TRaDe uses. This may be caused by the fact that AssureJ incorrectly removes information about dead threads but until they divulge their algorithm, it is anybody's guess.

6 Future Work

As can be seen from Table 2, the overhead of data race detection is still large when comparing to normal execution. We plan to further investigate whether it is possible to tighten the gap between normal execution of Java programs and our race detection through the use of a JIT compiler.

In TRaDe, we are using an advanced form of vector clocks, called 'accordion clocks', which dynamically grow and shrink. We will evaluate their performance in highly multi-threaded applications.

Recently, a static analysis technique, called 'escape analysis', has been applied to Java (see for example [1, 3, 18]). It is used to classify objects as escaping from a method or from a thread. Objects that

[6] This shortcoming was confirmed by their product support.

SwingSet Demo	A highly multi-threaded demo, included with the JDK 1.2, of the Swing widget set. It demonstrates buttons, sliders, text areas, ... The demo was run until it was fully loaded and displaying its initial screen. Immediately thereafter it was shut down.
Jess 6.01a1	Jess is a clone of the expert system shell CLIPS, rewritten entirely in Java. Input is provided so that is solves the famous 'The monkey and the banana' problem included with the Jess distribution. Run with: `jess.Main examples/fullmab.clp`
Resin 1.2b1	Resin is a web server entirely written in Java. It supports JSP, XML, JavaScript, XTP, XSL, ... The JSP files `hello.jsp`, `env.jsp`, `counter.jsp` and `index.jsp`, included in their examples directory, were requested in parallel.
Colt 1.0.1.56	Open Source Libraries for High Performance Scientific and Technical Computing in Java. A benchmark program is included in the distribution which does a number of matrix calculations. Run with: `cern.colt.matrix.bench.BenchmarkMatrix dgemm dense 10 2 0.99 false true 5 5 50 100 300 50 100`
Raja 0.4.0 pre2	Ray tracer in Java generating a picture demonstrating their Phong highlight capabilities. `raja.ui.Compute -v -p txt -r 128x128 -d20 -o Phong-128x128.png Phong.raj`

Table 1: Description of benchmark programs

	TRaDe		AssureJ		JProbe		Hotspot		Interpreter	
	s	MB	s	MB	s	MB	s	MB	s	MB
SwingSet	98.3	126.6	160.6	73.3	>1200 †	>650 †	20	41.8	15	29.8
Jess	370.3	12.1	610	17.5	>3600 †	>650 †	22	19.8	76	8
Resin	56.5	27.6	68	27.3	193	226.17	11.8	27.9	10	13.7
Colt	132.5	25	187.8	21.2	471.6	71	27.8	23.6	40.4	13.5
Raja	204.8	19.5	372	17.8	1945 †	1037 †	14.6	24.5	42.2	11.3

Table 2: Performance measurements

remain local to a method can be stack allocated, removing the overhead of heap allocation. Synchronisation operations can be removed when operating on objects that are local to a thread. Although this is a static, conservative technique, not applicable to Java programs dynamically loading classes, it would be interesting to have a clearer view on how many objects escape analysis can mark as thread local in comparison to TRaDe.

7 Conclusions

Data races are a serious problem inherent to multithreaded programs. There exist a number of general techniques to combat data races but they are slow and very intrusive. In this article, we have introduced a new technique, TRaDe, that uses information about the changing topology of the objects interconnection graph to more efficiently detect data races. Despite the fact that TRaDe does not cut corners while doing data race detection, it is faster than all known competition and comparable to the best in memory usage.

8 Acknowledgements

Mark Christiaens is supported by the IWT SEESCOA project (IWT/ADV/980.374) Our thanks go to Michiel Ronsse, with whom we had many stimulating discussions and to professor Jan Van Campenhout for many valuable corrections to this article.

References

[1] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Static Analysis Symposium 99*, September 1999.

[2] Ken Arnold and James Gosling. *The Java Programming Language.* The Java Series. Addison-Wesley, Reading, Massachusetts, USA, 1996.

[3] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronisation in Java . Technical report, Department of Computer Science, University of California, Santa Barbara, CA 93106, april 1999.

[4] Jong-Deok Choi, Barton P. Miller, and H. B. Netzer, Robert. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 1991.

[5] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 1–10, March 1990.

[6] C.J. Fidge. Partial orders for parallel debugging. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and distributed debugging*, pages 183–194, May 1988.

[7] Dieter Haban and Wolfgang Weigel. Global events and global breakpoints in distributed systems. In *21st Annual Hawaii International Conference on System Sciences*, volume II, pages 166–175. IEEE Computer Society, January 1988.

[8] Richard Jones and Rafael Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1996.

[9] KL Group, 260 King Street East, Toronto, Ontario, Canada. *Getting Started with JProbe.*

[10] Kuck & Associates, Inc., 1906 Fox Drive, Champaign, IL 61820-7345, USA. *Assure User's Manual*, 2.0 edition, march 1999.

[11] Doug Lea. *Concurrent Programming in Java.* The Java Series. Addison-Wesley, Reading, Massachusetts, USA, 1997.

[12] Time Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, Massachusetts, USA, second edition edition, April 1999.

[13] R. H. B. Netzer. *Race Condition Detection for Debugging Shared-Memory Parallel Programs.* PhD thesis, University of Wisconsin-Madison, 1991.

[14] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. In *Proceedings of the third ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 133–144.

[15] Robert H. B. Netzer and Barton P. Miller. Experience with techniques for refining data race detection. Technical Report CS-92-55, Dept. of Computer Science, Brown University, Department of Computer Science, Brown University, Providence, Rhode Island 02912, November 1992.

[16] Michiel Ronsse. *Racedetectie in Parallelle Programma's door Gecontroleerde Heruitvoering.* PhD thesis, Universiteit Gent, May 1999.

[17] Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[18] Erik Ruf. Effective synchronization removal for java. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 208–218. ACM, June 2000.

[19] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Operating Systems Review*, volume 31, pages 27–37. ACM, October 1997.

[20] Reinhard Schwarz and Friedman Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[21] Bill Venners. *Inside the Java Virtual Machine.* McGraw-Hill, New York, New York, USA, second edition, 1999.

The HotSpot™ Serviceability Agent: An out-of-process high level debugger for a Java™ virtual machine

Kenneth Russell[1] and Lars Bak[2]
Sun Microsystems
{Kenneth.Russell, Lars.Bak}@eng.sun.com

Abstract

The HotSpot™ Serviceability Agent (SA) is a set of APIs for the Java programming language which model the state of Sun Microsystems' Java™ HotSpot Virtual Machine. Unlike most previous debugging systems for dynamic languages which assume a "cooperative" model in which the target process runs code to assist in the debugging process, the SA requires no code to be run in the target VM. Instead, it uses primitives like symbol lookup and reading of process memory to implement its functionality. The SA can transparently examine either live processes or core files, making it suitable for debugging both the VM itself and Java programs in production. We describe the design and implementation of the SA, comparing it to other debugging systems for both statically compiled and dynamic languages, and illustrate future directions for this architecture.

Figure 1. An object inspector built with the SA's APIs.

1 Introduction

The Java HotSpot Virtual Machine implementation (hereafter referred to as the HotSpot JVM) is Sun Microsystems' high-performance VM for the Java platform[3]. HotSpot's origins are in language research in Beta [18], Smalltalk [6], and Self [12] [13]. The HotSpot JVM uses many advanced techniques to achieve high performance. The system employs a mixed-mode assembly interpreter which shares the stack with both C code and Java programming language methods (Java methods) compiled to machine code. Runtime profiling focuses compilation effort only on "hot" methods. Dynamic deoptimization [14] allows a compiled method to revert back to the interpreted state if invariants under which the method was compiled are compromised by future class loads. The ability to deoptimize allows the compiler to perform aggressive optimization and inlining.

How does one debug the product version of a highly-optimizing JVM which is written largely in C or C++ ? Such a JVM

- tends to operate with generated machine code, and to merge the C++ and Java virtual machine stacks (Java stacks). Activations on the stack corresponding to invocations of Java methods (Java frames) show up in a C++ debugger as raw program counters with no corresponding symbol.
- may have a highly-optimizing compiler. An activation on the stack may correspond to more than one Java method invocation, because of inlining.
- may encode many of its run-time data structures to save space.
- will not have available debug information for the C++ data structures describing portions of the run-time system such as the layout of the heap.

In short, when one examines the product version of such a JVM with a traditional C++ debugger, one deals with raw bits. All of the high-level abstract data types are gone.

[1]901 San Antonio Road, M/S UCUP02-302, Palo Alto, CA 94303.
[2]Computer Science Department, Aarhus University, Aabogade 34, DK-8200 Aarhus N, Denmark.
[3]An arbitrary Java virtual machine implementation is hereafter referred to as a JVM.

The HotSpot Serviceability Agent is a set of APIs for the Java programming language which recover this high-level state from a product-mode HotSpot JVM or a core file. Clients of the SA can use its APIs to write application-specific tools, debugging aids, and querying operations which operate directly on a target JVM and in a completely non-intrusive fashion; Figure 1 illustrates an object inspector built on top of the SA's APIs. Unlike most debuggers for dynamic languages, the SA requires no code to be run in the target JVM. It is therefore robust in the face of JVM failures. This property is what allows it to operate as a post-mortem JVM debugger. The SA is also applicable to more situations than just debugging the JVM; for example, end users can use it to write heap profilers which operate on servers in production without taking them down.

The rest of the paper is organized as follows. Section 2 compares the SA to other work in the field of debugging. Section 3 describes its contributions. Section 4 gives a concrete example of walking the JVM's thread list, and describes how the SA's APIs relate to the target JVM. Section 5 describes how the SA obtains type and offset information for C++ data structures in the target JVM. Section 6 details the low-level debugging primitives used by the SA. Section 7 explains the traversal of the remote JVM's heap, and gives an example of an object histogram. Section 8 describes how the SA walks the remote JVM's Java stacks. Section 9 discusses future work.

2 Related Work

We partition related work into two categories: debuggers for statically compiled languages (e.g., C, C++ , Fortran, Modula-2) and "dynamic languages", which are typically characterized by the ability to load and/or compile code at run time and which typically have a substantial run-time system including a garbage collector.

The two standard UNIX debuggers, dbx [17] and gdb [25], assume the model of an *uncooperative* debugging target. In this model it is assumed that the target process may be dead, in which case only examination-only operations are allowed. While both debuggers have a "call" facility which allows an arbitrary procedure to be executed in the target process if it is alive, a large percentage of the debugging tools are available for use on core files. These debuggers are language-independent, but have only been developed in the realm of statically-compiled languages like C, C++ , Fortran, and Modula-2.

Higher-level debugging tools for statically compiled languages have been developed as well. Sosic [23] describes a set of APIs which were used to implement debuggers in the Dynascope distributed system; these APIs map closely to much of the functionality provided in dbx and gdb. The notion of modeling the contents of the remote process's heap is introduced with an example of copying a linked list from one process to another assuming that the two processes are written in the same language. While language independence is claimed, no language other than C is described. Gough, Ledermann and Elms [11] describe a substantially different type of debugger which is targeted toward fully optimized machine code. Their system works in a "forward" fashion, using a modified compiler to evaluate expressions in the context of optimized code, rather than attempting to annotate the generated code with debugging information. This system also claims language independence, but was developed for statically-compiled languages like C, Pascal and Modula-2 [7]. Both of the systems described in Sosic and Gough et al. use an agent embedded in the target process to implement all of the functionality, thereby requiring that the target process be alive.

Debuggers for dynamic languages generally assume a *cooperative* debugging model; that is, the debugger cooperates with the execution environment (or virtual machine or "VM") for the language in order to allow debugging of programs written in the language. The debugger may be implemented in-process as in Perl [1], Python [29], and Scheme [2] or out-of-process using a wire protocol as in Spinellis' two-process Prolog debugger [24] or as in the Java Platform Debugger Architecture [19]. Some debuggers for safe languages like Tolmach and Appel's Standard ML debugger [27] and the Objective Caml debugger [8] take advantage of the safety of the target language to provide facilities not commonly found in debuggers for statically-compiled languages, such as the ability to step backward through the execution of the program. Gill's post-mortem debugger for Haskell [10] is in the same vein as the SA an examination-only system, but is post-mortem in the sense that the Haskell program has terminated abruptly, not the entire run-time system. All of these systems assume that code can be run in the target VM. Squeak [15] was developed and debugged using itself; the debugging methodology used was to prototype and debug in Smalltalk and then translate to C for high performance. Debugging a crashed Smalltalk VM is not discussed. Lisp systems, like the Allegro Common Lisp environment [9], often have the capability to dump an executable image. However, it appears that it is not possible to debug such a dumped image without running code in it.

The Jalapeño project has developed a JVM written in the Java programming language which is based around compiling Java bytecode to machine code [5]. Because nearly all of the code being executed is generated by the system itself, a traditional (C or C++) debugger is of no help in examining data structures. The debugging system for Jalapeño [21] uses the same mechanisms used by dbx, gdb, and the SA to attach to a target JVM and read its process memory without running code in the tar-

get. A unique aspect of the Jalapeño debugger is that since both the debugger and the JVM being debugged are written in Java, a modified Java interpreter can be used to execute the debugger's code which allows the JVM's data structures to debug themselves. This aspect of the Jalapeño debugger goes beyond what the SA can currently do, since the HotSpot JVM is implemented in C++.

3 Contributions

Compared to previous work, the HotSpot Serviceability Agent is closest in design to the Jalapeño debugger [21], which is also constructed around examination-only, out-of-process operations. In addition, the Jalapeño system, by taking advantage of the fact that the debugger and target JVM are implemented in the same language, achieves a level of elegance not currently possible with the SA.

The HotSpot Serviceability Agent was designed to be able to diagnose JVM failures. This requirement informed several design decisions, including that no code is run in the target process. The SA is currently an examination-only system, meaning that it derives all of its information using low-level primitives like symbol lookup and reading memory from a remote process. This allows it to work both by attaching to a running process as well as reading a core file. It can also run its code in an arbitrary JVM.

Compared to the Jalapeño work, this paper contributes a deeper discussion of debugging in the face of JVM failures, especially regarding heap and stack traversal. It describes a new architecture for interacting with remote Java objects which is similarly applicable to a run-time system. It discusses cross-language issues which arise when the target JVM is not implemented in Java. Finally, it analyzes the modeling process used to describe JVM data structures and discusses possible alternative implementations.

Because of the architectural similarities between the SA and the Jalapeño debugger, throughout this paper we explicitly point out some of the differences between the two systems. Chief among these is that the SA has been used to successfully diagnose difficult bugs in the production HotSpot JVM, whereas Jalapeño is a research system.

4 Example: Walking the Thread List

The HotSpot JVM maintains information indicating what kind of code each Java thread is executing: JVM-internal code, "native" code [16], or Java code. Consider the simple example of traversing the target JVM's thread list and determining this information.

Figure 2 provides a diagram of the interaction between the SA and the target JVM. The JVM uses the C++ data structure in 2A to describe the thread list. A static pointer in class `Threads` points to the head of the list. The memory layout of the resulting objects in the target JVM is shown in 2B. Figure 2C illustrates the analogous code from the SA. The SA does not actually use hard-coded offsets, as is described and illustrated later.

The APIs in the SA mirror the C++ structures. When an object created by the SA's code models an object in the target JVM, it fetches data from the target using the `Address` abstraction, which contains the illustrated methods as well as those which fetch strongly-typed, Java-sized primitive data like `byte getJByteAt(long offset)` and `short getJShortAt(long offset)`.

5 Describing C++ Types

In order to avoid hardcoded offsets in the SA's code (as used in Figure 2), it is necessary to model the types and structures of C++ objects in the target JVM, so that fields can be fetched by name. There are several ways that this structural information can be obtained, the most widely used and automatic being to have the C++ compiler generate debugging information such as *stabs* [17] during the build. These annotations are stored in the object file and resulting linked shared object or executable and can be parsed by the C++ debugger on the same platform.

The HotSpot JVM contains some classes which have fields defined to be the same sizes as some of the Java primitive types; for example, the C type `jshort`, defined by the JNI specification [16], is identical in size and signedness to the Java type `short`. In order to reduce the possibility of accidental errors in the SA's code, it was necessary to understand which fields in C++ types were actually "Java fields" and expose their contents as the correct Java primitive types in the SA.

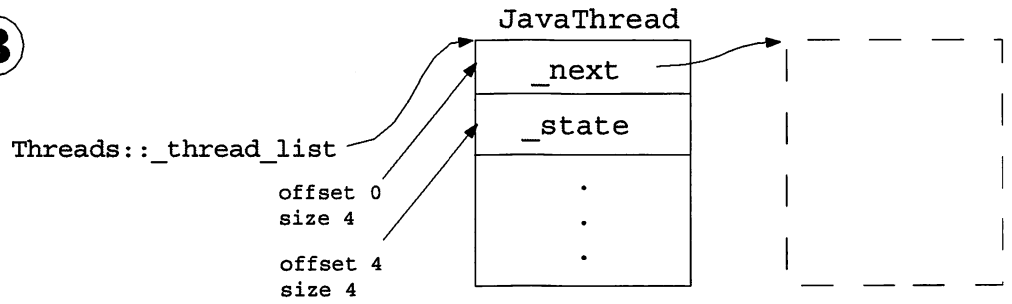
The nature of the process by which stabs and other debug information are generated typically loses such type information; for example, a `jshort` may be typedefed to a C `short int` or similar, and the debug information for a data structure containing a field of that type will identify the given field as a `short int`, not a `jshort`. This information loss was deemed unacceptable in the type modeling process for the SA.

For this reason, the structural information provided by the target JVM to the SA is presented in the form of three tables which are generated by C++ preprocessor macros and compiled into the target JVM. We illustrate only the first of these tables in Figure 3A. This table contains name, type, and offset or address information for nonstatic and static fields, respectively. The second table models the C++ inheritance hierarchy, and the third provides named integer constants needed by the SA to properly traverse data structures. The SA uses a general symbol lookup mechanism to locate these tables and

(A)

```
class JavaThread {
  JavaThread* _next;
  // _in_vm, _in_native, _in_Java
  JavaThreadState _state;
public:
  JavaThread* next()       { return _next; }
  JavaThreadState state()  { return _state; }
};
```

(B)



(C)

```
public class JavaThread {
  private Address addr;
  public JavaThread(Address addr) { this.addr = addr; }
  public int state() {
    // offset, size, isUnsigned
    return (int) addr.getCIntegerAt(4, 4, false);
  }
  public JavaThread next() {
    Address nextAddr = addr.getAddressAt(0);
    if (nextAddr == null) return null;
    else return new JavaThread(nextAddr);
  }
}
```

Figure 2. Illustration of the mirroring of the JVM's data structures in the SA. (A) shows a subset of the JVM's JavaThread code, including the state of the thread (in JVM code, in native code, or in Java code) and the structure of the thread list. (B) illustrates the memory layout of this data structure in the JVM's address space; starting with the global thread list, JavaThread objects are linked together. (C) shows the SA code which accesses this data structure. Access to the start of the list is not shown.

(A)

```
typedef struct {
  const char* typeName;   // Type name containing field (Example: "JavaThread")
  const char* fieldName;  // Field name within type (Example: "_state")
  const char* typeString; // Quoted type of field (Example: "JavaThreadState")
  int32_t  isStatic;      // Effectively a boolean
  uint64_t offset;        // Used for nonstatic fields
  void* address;          // Used for static fields
} VMStructEntry;

extern "C" VMStructEntry* gHotSpotVMStructs;  // The table itself
// The following offsets describe the layout of the table for bootstrapping
extern "C" uint64_t gHotSpotVMStructEntryTypeNameOffset;
extern "C" uint64_t gHotSpotVMStructEntryFieldNameOffset;
...
```

(B)

```
public interface TypeDataBase {
  public Type lookupType(String name);
}

public interface Type {
  // Java fields:
  public JByteField getJByteField(String fieldName)
    throws WrongTypeException;
  ...
  // C fields:
  public CIntegerField getCIntegerField ... ;
  public AddressField getAddressField ... ;
}
```

(C)

```
public class JavaThread {
  private Address addr;
  private CIntegerField stateField;
  private AddressField  nextField;
  static {
    TypeDataBase db = VM.getVM().getTypeDataBase();
    Type type  = db.lookupType("JavaThread");
    stateField = type.getCIntegerField("_state");
    nextField  = type.getAddressField("_next");
  }

  public JavaThread(Address addr) { this.addr = addr; }
  public int state() {
    return (int) stateField.getValue(addr);
  }
  public JavaThread next() {
    Address nextAddr = nextField.getValue(addr);
    if (nextAddr == null) return null;
    else return new JavaThread(nextAddr);
  }
}
```

Figure 3. Modeling of C++ types in the SA. (A) shows the structure of one of the JVM's three type tables and the global symbols used to access and describe it. (B) highlights the salient aspects of the TypeDataBase and Type interfaces. (C) shows a portion of the actual SA code; compare to the hardcoded offsets in Figure 2.

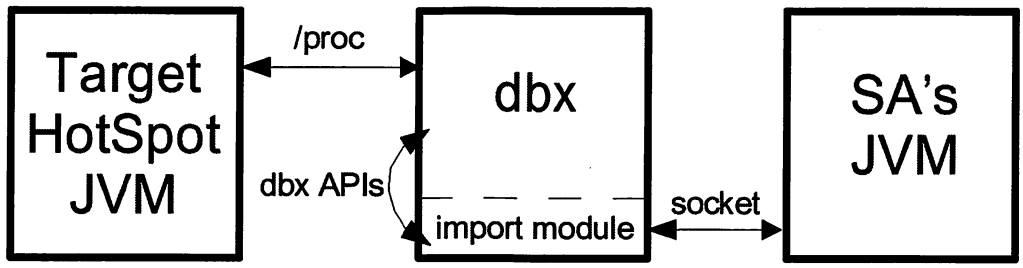


Figure 4. Block diagram of connections between the SA and the target HotSpot JVM on Solaris.

parses them upon attaching to the target JVM or core file, effectively downloading the information it needs to analyze the remote JVM.

This approach has two additional advantages over using the compiler's raw debug information: first, since the symbols being referenced in the SA are explicitly exported by the JVM, changes to the JVM code which are incompatible with the SA's code are often encountered at build time as a failure during compilation of the side tables. Second, it reduces porting time by eliminating the need to understand the platform-specific debugging information generated by the C++ compiler.

The structural information for C++ types is stored in a `TypeDataBase` (Figure 3B), which supports looking up a `Type` by name. From the `Type`, `Fields` can be fetched by name. The `Field` mechanism provides type checking between the SA's code and the target JVM, in addition to ensuring that field offsets are always kept synchronized as in Figure 3C.

Because the SA models C++ types as first-class Java objects, the relatively simple current implementation of the type system can be straightforwardly extended in the future.

6 Access to the Remote Process

The SA is built on top of a few very simple debugging primitives, including

- attach to remote process or core file
- lookup symbol in remote process
- read remote process memory

The Solaris™ Operating Environment (hereafter Solaris) version of the SA uses the native debugger, dbx, to obtain this functionality, encapsulating it in a small interface called `Debugger`.

When the SA is launched, it uses `java.lang.Runtime.exec()` to launch a subordinate dbx process. It sends commands to dbx to load and execute a small piece of self-contained C++ code, called an *import module* (Figure 4), and either to attach to the target HotSpot JVM (causing it to be suspended, as with any program being debugged) or to load a core file. The import module communicates with dbx via a small set of internal APIs and with the SA using a custom text-based protocol over a socket, and provides the SA with the above debugging primitives. The SA itself is written entirely in Java.

The only significant additional mechanism that has been added to the SA's `Debugger` interface as the system has evolved is the ability to query the CPU register set for a given thread. The need for this is described in Section 8.

Compared to the Jalapeño debugger, the SA explicitly models the interaction with the remote address space. This was a requirement since the target JVM was not implemented in Java and, therefore, JVM data structures could not be directly repurposed (and still write the SA in Java). Explicitly describing the interaction with the remote address space conveys two additional advantages: first, it describes failures using the Java language (for example, by throwing an `UnmappedAddressException`), and thereby allows the debugging system to become more robust where necessary: see Section 7. Second, it opens up the possibility for language-independent debuggers written in Java; for example, the SA's modeling of C++ types is the first small step toward writing a C++ debugger. This is discussed further in Section 9.

7 Traversing the Heap

The SA exposes HotSpot's generational [28] garbage collection framework in an abstraction called the `ObjectHeap`. The most significant operation provided by this abstraction is the ability to uniformly visit all fields in all objects in the heap. These objects include both Java objects as well as JVM-internal reflective objects like methods and classes. For this reason objects are termed `Oops`, a term borrowed from the Smalltalk community.

Figure 5 illustrates the iteration mechanism provided to clients. Implementing debugging and profiling tools as shown in Figures 1 and 6 requires very little additional code.

The iteration mechanism can be made robust in the face of JVM failures. Consider the situation where a bad

```
// An OopVisitor can be used to inspect
// all fields within an object.
// Fields include vm fields, java
// fields, indexable fields.

public interface OopVisitor {
  // Called before visiting an object
  public void prologue();

  // Called after visiting an object
  public void epilogue();

  public void setObj(Oop obj);

  // Returns the object being visited
  public Oop getObj();

  // Callback methods for each field type
  // in an object
  public void doOop(OopField field);
  public void doByte(ByteField field);
  public void doChar(CharField field);
  public void doBoolean(BooleanField field);
  public void doShort(ShortField field);
  public void doInt(IntField field);
  public void doLong(LongField field);
  public void doFloat(FloatField field);
  public void doDouble(DoubleField field);
  public void doCInt(CIntField field);
}
```

Figure 5. The OopVisitor interface, which provides uniform iteration over all objects in the heap.

pointer has been stored in an object field of a Java object. Constructing an `Oop` for the value stored in that field will likely result in an `UnmappedAddressException` or `UnalignedAddressException`. Client code, for example in an object inspector, can explicitly catch these exceptions and raise a red flag in the user interface if one arises. It is not clear from the description of the Jalapeño debugger how similar functionality is provided, since it seems that such exceptions are not modeled explicitly in the Java language, but instead handled somehow by the underlying remote-aware interpreter.

8 Traversing the Stacks

A debugger is hardly useful if it does not provide detailed information about where the program has failed. In the context of debugging a static program, for example from a core file, this means providing a stack backtrace of all threads. Figure 7 shows the trace for one thread.

The HotSpot JVM necessarily has internal abstractions for describing and walking the stack; it performs type-precise garbage collection [4], and must therefore be able to walk the stacks of all Java threads at GC time, precisely locating all pointers to live Java objects and updating those pointers if the object is moved during GC.

The HotSpot JVM uses a safepointing mechanism [3] to halt execution of interpreted or compiled Java code only when the locations of all objects are known precisely. (Recent work has described how to eliminate the need for safepoints for the purposes of garbage collection [26], but HotSpot also uses safepoints for other system-wide operations such as deoptimization.) Metadata generated by HotSpot's compilers describes the state of compiled code in great detail at safepoints, including any inlining the compiler may have performed.

The code in HotSpot's run-time system was designed to operate at safepoints. In a debugging scenario, however, the JVM will in general not be at a safepoint; it may be suspended, or may have crashed, at an arbitrary point in its execution.

The SA's stackwalking code was ported from the JVM and modified over time to handle problems which arise when the JVM is not at a safepoint:

- the JVM may not have metadata for a program counter in compiled Java code. In this case the metadata associated with the closest possible PC is used, unless there is no such information (as in a leaf method), in which case the iteration code assumes that there is, for example, no inlining.
- an interpreter frame may not have been set up yet, yielding an incorrect bytecode index. In this case the iteration code skips the topmost frame on the stack.
- a thread executing generated machine code may have been interrupted by a signal.
- the topmost Java frame may not be available to the run-time system at all.

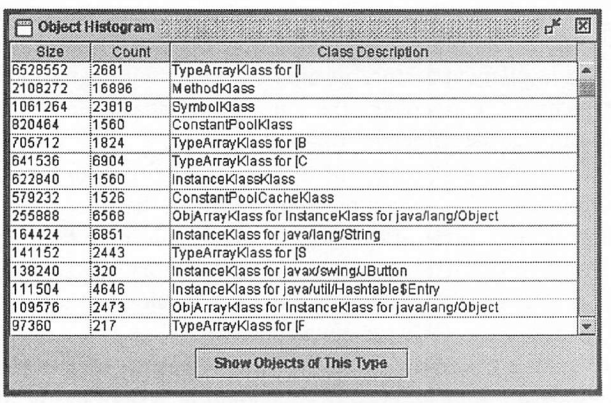


Figure 6. Histogram of objects in the heap.

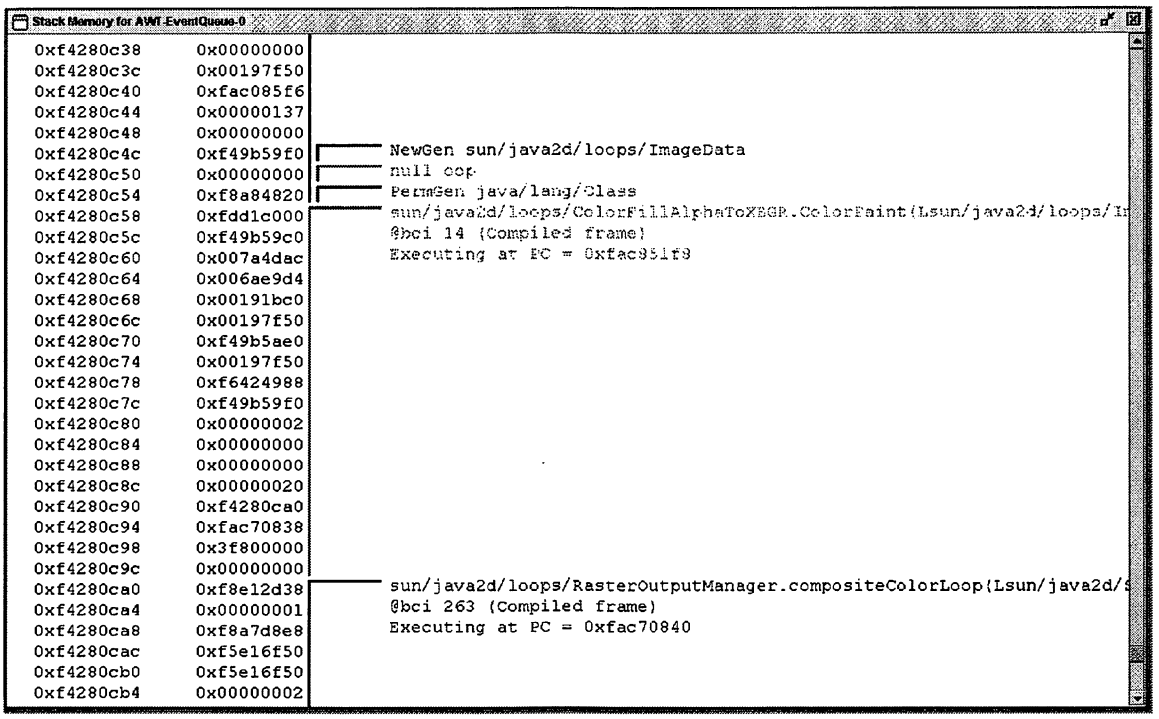


Figure 7. Stack memory annotated with Java frames and live objects.

The latter problem is the most significant. When the JVM reaches a safepoint, each thread currently executing Java code enters the run-time system, storing its last known Java stack pointer into thread-local storage in preparation for stack traversal. The JVM's run-time code traverses only Java frames, skipping around sequences of C frames. In a debugging scenario the entire topmost sequence of Java frames for a given thread will be unreachable with the JVM's built-in stackwalking code if the thread was suspended (or crashed) while executing either interpreted or compiled Java code.

The SA solves this problem by using the `Debugger` interface to fetch the last known stack pointer for a given thread (Section 6). Given that stack pointer, it must be able to walk backward to find the first Java frame on the stack, proceeding as usual from that point on. We have found that the JVM's run-time code is basically adequate to handle this technique on the SPARC architecture, but expect that in a forthcoming port to the x86 we will have to implement a more general stackwalking mechanism to handle cases where the frame pointer has been eliminated, as described by Linton [17].

A full description of the technique for recognizing and traversing signal handler frames on the stack is beyond the scope of this paper. We briefly note that we have implemented a not-quite-satisfactory mechanism on the SPARC architecture as an extension to the SA's port of the JVM's run-time code.

Our experience has been that implementing stackwalking in the context of a debugging system has been non-trivial. The Jalapeño debugger rightly emphasizes its reuse of JVM code in examining the remote JVM's data structures. We have found, however, that in order to handle and diagnose JVM failures, significant divergence from the JVM's code is required in the area of stackwalking.

9 Discussion and Future Work

The Serviceability Agent is similar in architecture to the Jalapeño debugger. The most interesting aspect of the latter is that the same code can both implement and debug the JVM's run-time system. A valid criticism of the SA is that it requires duplication of JVM code, since the SA and JVM are implemented in different languages (and necessarily with different underlying architectures). We have found a few unexpected instances of version skew which silently broke the SA. For the most part, however, the table-based symbol export mechanism described in Section 5 provides early warning when the SA will break. Roughly 13,000 lines of the SA's 35,000 lines of code must track the VM with varying degrees of closeness.

Compared to other previous work, the SA appears to be the first system which has the ability to debug in a high-level manner data structures from both static languages like C++ and dynamic languages like Java. Other dynamic languages could be targeted by the SA by using its debugging primitives to model the run-time system of the target VM, as has been done for HotSpot. This is

currently a labor-intensive process but has the advantage of providing a true post-mortem debugger.

The SA operates either on a JVM that has been suspended by platform-specific debugging mechanisms or on a core file. Solaris contains a program called `gcore` which takes a core file snapshot of a running process without terminating that process; it is frequently used in conjunction with dbx to examine the state of production systems written in C or C++ rather than attaching a debugger directly [22]. The SA allows gcore to be used for Java programs as well. The speed of the SA's access to the target JVM's heap is substantially slower than that enabled by the current standard API, the Java Virtual Machine Profiling Interface [20]. However, JVMPI imposes a significant performance overhead to the normal running of the application. Used in conjunction with gcore, the SA only has the impact of suspending the process while taking the snapshot, which takes a few seconds for programs of normal heap size.

An early design decision was to use only examination-only debugging primitives, to make the goal of the system clear and to explore what was possible to build using them. If the target JVM is alive (not a core file) and running properly, being able to write to the process memory would facilitate more rich interactions when debugging. Some of the possible features, like setting fields, would be straightforward to implement and would not require JVM code changes, while others, like setting breakpoints, would likely require a substantial amount of supporting code in the JVM. An advantage of the SA's architecture is that it could degrade gracefully back to an examination-only system if the JVM crashed.

In the future, we plan to complete support for Solaris/x86, Linux, and Win32. We will continue to expand the SA's knowledge of JVM data structures. We plan to write more tools using the SA's APIs, and to explore the addition of debugging primitives which affect the state of the target JVM. We will investigate if it would make sense from a performance or footprint standpoint to rewrite portions of the JVM's run-time system in Java.

10 Acknowledgments

We thank Ivan Soleimanipour and Siva Annamalai for their help with the system's use of dbx; our managers, Jerry Driscoll and Tricia Jordan, for their support of the project, especially in its early stages; the entire HotSpot group for encouragement; and Robert Griesemer in particular for reviewing early drafts of this paper. We also thank the anonymous reviewers for their helpful comments.

References

[1] CPAN: comprehensive perl archive network, 2000. `http://www.cpan.org/`.

[2] The Scheme programming language, 2000. `http://www.swiss.ai.mit.edu/projects/scheme/index.html`.

[3] O. Agesen. GC points in a threaded environment. Technical Report SMLI-TR-98-70, Sun Microsystems Laboratories, December 1998.

[4] O. Agesen, D. Detlefs, and J. E. B. Moss. Garbage collection and local variable type-precision and liveness in Java Virtual Machines. *ACM SIGPLAN Notices*, 33(5):269–279, 1998.

[5] B. Alpern, D. Attanasio, J. Barton, A. Cocchi, S. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, Denver, Colorado, November 1 1999. `http://www.research.ibm.com/jalapeno/publication.html#oopsla99_jvm`.

[6] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference record of the 11th ACM Symposium on Principles of Programming Languages (POPL)*, pages 297–302, 1984.

[7] K. Elms. Debugging optimised code using function interpretation. In Mariam Kamkar, editor, *AADEBUG '97, Proceedings of the Third International Workshop on Automatic Debugging*, pages 27–36, Linköping, Sweden, May 26-27 1997.

[8] X. Leroy et al. The Objective Caml system release 3.00, 2000. `http://caml.inria.fr/ocaml/htmlman/`.

[9] Franz Incorporated. Online documentation in html as part of the Allegro Common Lisp 5, July 1998.

[10] A. Gill. Haskell object observation debugger, 2000. `http://www.haskell.org/hood/`.

[11] K. Gough, J. Ledermann, and K. Elms. Interpretive debugging of optimised code, 1994.

[12] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Proceedings ECOOP'91*, pages 21–38, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[13] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI '94 Conference Proceedings*, pages 326–335, Orlando, FL, June 1994.

[14] U. Hözle, C. Chambers, and D. Ungar. Debugging optimised code with dynamic deoptimization. In *Proceedings of the SIGPLAN '92 Conference on Programming Languages Design and Implementation, SIGPLAN Notices*, pages 32–43, San Francisco, CA, June 1992. ACM Press.

[15] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *OOPSLA'97 Conference Proceedings*, pages 318–326, 1997.

[16] Sheng Liang. *The Java Native Interface*. Addison-Wesley, Reading, MA, 1999.

[17] M. A. Linton. The evolution of dbx. In *Proceedings of the 1990 Usenix Summer Conference, Anaheim, CA*, 1990.

[18] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, 1993.

[19] Sun Microsystems. Java platform debugger architecture, 2000. `http://java.sun.com/j2se/1.3/docs/guide/jpda/`.

[20] Sun Microsystems. Java virtual machine profiling interface, 2000. `http://java.sun.com/j2se/1.3/docs/guide/jvmpi/`.

[21] T. Ngo and J. Barton. Debugging by remote reflection. In *Proceedings of Euro-Par 2000*, Munich, Germany, August 27-September 1 2000. `http://www.research.ibm.com/jalapeno/publication.html#remotereflect`.

[22] I. Soleimanipour and S. Annamalai. Personal communication, 2000.

[23] R. Sosic. A procedural interface for program directing. *Software: Practice and Experience*, 25(7):767–787, 1995.

[24] D. Spinellis. The design and implementation of a two process Prolog debugger. Technical Report IR-LP-31-21, ECRC, September 1989.

[25] R. Stallman and R. Pesch. Debugging with GDB: The GNU source-level debugger, 1999.

[26] J. Stichnoth, G. Lueh, and M. Cierniak. Support for garbage collection at every instruction in a Java compiler, 1999.

[27] A. P. Tolmach and A. W. Appel. Debugging Standard ML without reverse engineering. In *ACM Conference on Lisp and Functional Programming*, pages 1–12, 1995.

[28] D. M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, 1984.

[29] G. van Rossum et al. Python library reference: The Python debugger, 2000. `http://www.python.org/doc/current/lib/module-pdb.html`.

More Efficient Network Class Loading through Bundling

David Hovemeyer and William Pugh
Department of Computer Science
University of Maryland
daveho@cs.umd.edu, pugh@cs.umd.edu

Abstract

In this paper, we describe *bundling*, a technique for the transfer of files over a network. The goal of bundling is to group together files that tend to be needed in the same program execution and that are loaded close together. We describe an algorithm for dividing a collection of files into bundles based on profiles of file-loading behavior.

Our motivation for bundling is to improve the performance of network class loading in Java. We examine other network class loading mechanisms and discuss their performance tradeoffs. Bundling is able to combine the strengths of both on-demand strategies and archive-based strategies. We present experimental results that show how bundling can perform well in a variety of network conditions.

Introduction

Through *class loaders*, the Java[1] virtual machine (JVM) supports a flexible model for loading executable code and resources at runtime [8]. *Network class loading* is an important form of class loading, in which class files and resources are transferred to the client JVM over a network from a server.

Because networks are generally slower in bandwidth and latency than local disks, users can experience poor startup times and significant delays when running an application loaded from a network. It is therefore desirable to optimize the transfer of classes and resources over the network to minimize these delays. Specifically, we would like to find an approach that has the following properties:

- Only files needed by the client are sent over the network.
- Files arrive when needed, in the order they are needed.
- As few bytes as possible are transferred.
- The number of requests made to the server should be minimized, in order to reduce the delay due to network latency.

In addition, we would like to make the approach practical for 'real-world' deployment:

- The server implementation should scale to a large number of clients.
- The client code should be small and efficient, and should use only standard parts of the Java runtime (i.e., no native code).
- The client code should have minimal overhead.

This paper describes a technique called *bundling* that can approximate all of these properties under realistic network conditions.

The document is structured as follows. Section 1 reviews existing network class loading mechanisms. Section 2 describes bundling. Section 3 presents some experimental results comparing the effectiveness of bundling to existing network class loading mechanisms. Section 4 describes related work. Finally, section 5 summarizes the contributions of this paper and suggests possibilities for future work.

1 Overview of mechanisms for network class loading

This section discusses issues encountered in network class loading, and how these issues are handled by existing technologies for network class loading in Java.

1.1 Issues in network class loading

We identified several design issues that must be addressed in any network class loading technique. This section discusses these design issues and some of the tradeoffs they imply.

One network connection *vs.* multiple. When the client requests class files and resources from the server, it could use a separate network connection for each file, it could use a single persistent network connection for all files, or it could use a fixed number of persistent connections. Using a single persistent connection is generally the best approach. Using separate connections for each file is generally not a good idea since setting up a network connection is expensive. It is not clear that there would be any advantage to using multiple persistent connections, since in class loading there is no obvious advantage to downloading files in parallel.

Transfer granularity: individual files *vs.* many. The ability to send individual files is desirable because it allows the files sent by the server to exactly match those needed by the client. As the granularity of the transfer units increases, the probability that unneeded files might be sent increases, as does the probability that files will be sent in an order different from the request order. However, small transfer granularity increases the likelihood of of delays due to network latency, because the client must send requests to the server more frequently.

Using compression *vs.* not. Compressing class files and resources is desirable when network bandwidth is limited, since it results in transferring fewer bytes over the network. However, the benefit of transferring fewer bytes must be weighed against the runtime cost of performing the compression and decompression.

Compressing individual files *vs.* multiple. Applying compression to multiple files is desirable since it offers more opportunities for sharing redundant information between files, thus increasing the compression ratio.

Performing compression on-line *vs.* off-line. Performing compression off-line is desirable because it reduces the amount of computation required on the server.

Pre-sending/prefetching *vs.* not. When classes and resources are transferred on-demand, the client must send requests for the files it needs to the server. If files are requested only at the precise point they are needed, then client must pay the cost of network latency for each request. This latency cost can be reduced or eliminated if files are either pre-sent or prefetched prior to the point when they are needed by the client. However, care must be taken that files are not pre-sent or prefetched unnecessarily.

1.2 Existing mechanisms for network class loading

This section describes existing mechanisms for network class loading, and how they address the issues described in section 1.1.

1.2.1 Downloading individual files

One of the standard network class loading mechanisms in Java is for the client to request files individually relative to a directory URL, typically using the HTTP protocol.

Downloading individual files has the advantage that only those files explicitly requested by the client are transferred over the network. This ensures that no unneeded files are transferred, and that the files arrive in the order in which they are needed. HTTP connections are usually persistent, meaning that a single connection is used to transfer many files, and that the connection persists between client requests. In principle, it is also possible for the HTTP client and server to negotiate the use of compression; however, such compression is per-file and does not share redundant information between files. (In any case, the JDK 1.2.2 implementation of HTTP does not support such compression.)

Downloading individual files has the disadvantage that files are not pre-sent or prefetched, meaning that the network latency cost is paid for each file requested by the client. Given that network latency can be hundreds or even thousands of milliseconds, this disadvantage can be considerable.

1.2.2 Downloading Jar archives

Another standard network class loading mechanism in Java is to download Jar archive files [10]. Jar archives use the same underlying format as Zip files [5]. Each file in the archive is usually (though not always) compressed using the Deflate algorithm [4].

Jar files are convenient for application developers because they provide a simple mechanism for packaging an entire application or a component of an application into a single file. Since Jar files contain multiple files, they can be considered a form of pre-sending by the server, meaning that the network latency cost of the request is paid only once, for the entire Jar file, rather than once for each class or resource file. Jar files also offer random access to the files contained in the archive, which is useful for loading classes and resources from a Jar archive on disk. However, when used to transport classes and resources over a network they are likely to deliver unneeded files or to deliver files in the wrong order, because they typically contain a large number of files. In addition, the fact that the files are compressed individually results in a

lower compression ratio than would be possible if the entire archive were compressed using cumulative compression.

JDK version 1.3 introduced *Jar indexes*. A Jar index is a mapping from class and resource files to the names of the Jar files in which they are located. The Jar index for a set of Jar files is contained in a *master Jar file*. Once the master Jar file and the Jar index are received by the JVM, subsequent Jar files are downloaded only if they contain a class or resource file needed by the running application. This feature is an improvement over the technique used by earlier JDK versions to find class and resource files in a collection of Jar files: each Jar file was simply downloaded in sequence until the desired file was located. Through the use of a Jar index, an application can be split up into discrete components which are downloaded only when needed. This can help reduce the number of unneeded classes and resources downloaded. (The functionality of the Jar index is in some ways similar to our concept of *bundling*, which will be described in section 2.)

1.2.3 On-the-fly compression

On-the-fly compression is a variation on downloading individual files. Files are requested individually by the client, and transferred from the server over a compressed stream. Any suitable compression algorithm could be used; zlib [1] is an obvious choice since it is a standard part of the Java API (in the `java.util.zip` package)[2].

On-the-fly compression has the advantages of downloading individual files: only the files requested by the client are transferred, and the files arrive in the order needed. The fact that the entire stream of files is compressed cumulatively means that a high degree of sharing of redundant information between files is possible, resulting in compression ratios higher than those achieved by simply compressing each file individually.

However, on-the-fly compression has the disadvantage that it can require considerable CPU time on the server to perform the compression. For example, a 333 MHz Sun Ultra 5 can compress about 2.5 megabytes per second using zlib at the lowest compression level, and about 0.5 megabytes per second at the highest compression level. Considering that server will have more work to do at runtime in addition to compression, the sustainable throughput will be somewhat less. This limits the scalability of on-the-fly compression using zlib.

Another disadvantage of on-the-fly compression is that it does not use pre-sending or prefetching, meaning that the network latency cost is paid for each file requested.

1.2.4 Pack

Pack is a custom archive format for Java class files, developed by William Pugh [9]. It exploits regularities in the Java class file format to achieve a high degree of compression, and is designed as an alternative to Jar files. Like Jar files, the Pack format has the advantage that a large number of files can be delivered in response to a single client request, meaning that the request latency cost is paid only once for the entire archive, rather than per-file. It shares the disadvantage that unneeded files may be sent, and that the files may not arrive in the correct order.

Pack must be downloaded prior to use. The decompressor requires 36 kilobytes when downloaded as a Jar file.

Pack's decompressor is slower than zlib's. On a 333 MHz Sun Ultra 5 workstation it can decompress about 75-120 kilobytes per second, limiting Pack's effectiveness for fast networks.

1.3 Comparison of existing network class loading mechanisms

Table 1 summarizes existing network class loading mechanisms in terms of how they address the design issues discussed in section 1.1, along with how those issues would be addressed by an 'ideal' network class loading mechanism. None of the existing mechanisms has all of the properties we would like in an ideal mechanism. The request granularity of the archive formats (Jar, Pack) is larger than we would like. On-the-fly compression has the desired request granularity, but requires the compression to be performed at runtime, and does not use pre-sending or prefetching to reduce latency costs.

Jar indexes offer an intriguing possibility: we could break the application into chunks, put each chunk into a separate Jar file, and use the Jar index to inform the JVM which class and resource files are contained in each chunk. This scheme has some advantages over using a single monolithic Jar archive. If we choose the division of files into chunks carefully, so that only files needed together are put in the same chunk, then we can avoid downloading files that are not needed. We can also try to order the files within the chunks such that they match the client's request order at runtime. However, this scheme has some undesirable properties. If we want to allow a truly arbitrary mapping of files to chunks, the size of the Jar index will be proportional to the number of files. Also, we are still left with the problem that the chunks are encoded as Jar files, so the files within the chunks are compressed individually, not cumulatively.

	number of net connections	request granularity	compress?	compress scope	compress time	pre-send/ prefetch?
'Ideal'	1	individual file	yes	all files	off-line	yes
Individual files	1[3]	individual file	maybe[4]	individual file	off-line?	no
Jar archive	1	archive[5]	usually	individual file	off-line	yes
On-the-fly	1	individual file	yes	all files	runtime	no
Pack	1	archive	yes	all files	off-line	yes

Table 1: Summary of existing network class loading mechanisms and an 'ideal' network class loading mechanism.

2 Bundling: a hybrid approach

Bundling is an approach to transferring files over a network that tries to combine the benefits of individual file downloading, Jar file downloading, and on-the-fly compression. The collection of files comprising the application is divided into groups, or *bundles*. Each bundle is then compressed cumulatively using either a general-purpose compression mechanism such as zlib, or a Java-specific archive format such as Pack.

As in individual file downloading, bundles are transferred in response to explicit client requests. Ideally, each bundle will consist entirely of files that are always loaded together. In addition, the files in the bundle should be ordered such that they match the order of requests by the client.

As in Jar file downloading, the bundles are precompressed, so no compression needs to be performed on the server at runtime. This allows the server to scale more easily to a large number of clients.

As in on-the-fly compression (and unlike Jar files), compression is performed on multiple files instead on individual files. This allows more opportunities to share redundant information between files, resulting in a better compression ratio than with individual compression.

2.1 Dividing a collection of files into bundles

The main problem in applying bundling to network class loading is determining how to divide the collection of files needed by applications into bundles. In order to ensure that the client generally receives only files it needs, we must only put two files in the same bundle if those files are always (or almost always) needed together in the same program execution. Furthermore, the files in the bundle should always (or almost always) be ordered in the same order that they will be requested by the client. When these two properties hold, bundling has all of the desirable properties of on-demand class loading, with the additional benefits of compression and pre-sending.

Clearly, to achieve a good division of files into bundles, we need to have knowledge of a typical program's class and resource loading behavior at runtime. We chose to use *class loading profiles* as the source of information about program behavior. Class loading profiles record the order and time at which each class or resource was loaded during execution.

2.2 Conceptual framework

We chose to view the collection of files as a fully connected weighted graph. Each file is represented by a node in the graph. The weight of an edge from file A to file B represents the desirability of placing A and B in the same bundle.

Given a set of profiles, there are many ways to determine the edge weights. We chose to use *frequency correlation* as the basis for the edge weights. The frequency correlation of two files A and B is defined as t/n, where t is the number of profiles in which both A and B are loaded, and n is the number of profiles in which either A or B is loaded. A frequency correlation of 1.0 indicates that A and B are always loaded together.

The bundling algorithm considers edges according to an order produced by the *edge comparator*. We chose to use weight as the primary criterion for the edge comparator, and average distance as the secondary criterion. (The average distance of an edge connecting files A and B is the average distance between A and B in the input profiles.) This sort order helps ensure that edges connecting files usually loaded close to each other are considered before edges connecting files usually loaded far apart.

While frequency correlation is a good measure of how often two files are loaded together, it is not a measure of whether those files are generally loaded near each other in the profiles. If we put two files which are generally loaded far apart from each other in the same bundle, if a request is made for one of the files the other will be loaded too early. To ensure that all of the files in a bundle are loaded near each other, we limit the maximum *bundle spread*. The bundle spread for a proposed bundle b is the

maximum over all profiles p of

$$\text{lastMoment}(b,p) - \text{firstMoment}(b,p) - \text{size}(b) + 1$$

The 'moment' of a file in a profile is the ordinal value indicating when it was loaded relative to the other files in the profile. (I.e., the first file in the profile is moment 0, the second is moment 1, etc.) The best possible spread for a bundle is 0, indicating that for any profile in the input set, after any file in the bundle is requested all of the files in the bundle will be used before any file not in the bundle.

In addition to determining which files to bundle together, the bundling algorithm must also determine the order of the files within each bundle. This order is determined by a *bundle sort comparator*. We chose to use *average position* as the criterion for the bundle sort comparator. Given a bundle b and a profile p, the position of each file is found in relation to the other files in b. For example, if A is loaded in p before any other files in b, then its position is 0. The average position of file A in a bundle b is simply its average position over all input profiles. Using average position as the criterion for the bundle sort comparator helps ensure that files are placed in the order in which they will be needed, based on the order in which they occurred in the input profiles.

2.3 Bundling algorithm

Once the input profiles have been used to create the graph, the bundling algorithm uses the graph to put the files into bundles. The algorithm works as follows:

1. Initially, each file is put in a separate bundle.
2. Edges whose weight is less than the *minimum edge weight* are discarded.
3. The edges are sorted according to the *edge comparator*.
4. The edges are processed one at a time, in the order determined in step 3. For each edge, if the files connected by the edge are not already in the same bundle, and if the number of files in the resulting bundle would not exceed the *maximum bundle size*, and if the resulting bundle would not exceed the *maximum bundle spread*, then the bundles containing the files are combined into a single bundle.
5. After all the edges have been processed, the files within each resulting bundle are sorted according to the *bundle sort comparator*.

Once the contents of the bundles have been determined, they are combined and compressed cumulatively. We evaluated both zlib (at the highest compression level) and Pack as compressed formats for the bundles.

2.4 Implementation

The bundle class loader consists of a *server* and a *client*.

The server accepts requests for files from the client. For each request, it sends the bundle containing the requested file. Because the bundles are pre-compressed, the server does not perform any compression at runtime. As a result, the server has little CPU overhead.

The client consists of an implementation of `java.lang.ClassLoader`. When a call to `loadClass()` or `loadResource()` is made to the class loader, it first checks to see if the data for the requested class or resource has already been received. If so, it handles the request using the data already received. Otherwise, it sends a request for the needed file to the server. In return, it receives a bundle from the server containing the requested file. Because the bundle generally contains files not explicitly requested by the client, the client caches the data for all files received in memory. Once the data for a class or resource has been used by the client, its cache entry is deleted, so that the memory can be reclaimed by the garbage collector.

The implementation of the client and server are written in Java, consisting of 733 lines of source overall. The total size of the client class files is approximately 20 KB. Both client and server use only the standard Java runtime classes in their implementation, and will work with any JVM compatible with Sun's JDK 1.2.

Our implementation currently supports only bundles compressed with zlib. Future work will add support for bundles in Pack format.

2.5 Latency issues

Network latency has a substantial effect on the performance of any request-driven file transfer protocol, since it causes a delay between the time the client issues a request for a file and the time when the data for that file starts to arrive. This is one of the principal disadvantages of purely on-demand transfer strategies, such as downloading individual files. The request latency problem is compounded when the available bandwidth is high, since the penalty associated with each request is the product of the latency and the bandwidth (since the product is the amount of data that can be transferred during one latency period).

To decrease the penalty associated with network latency, it is necessary to reduce the number of requests issued by the client. The simplest way to do this in bundling is to find ways to increase the bundle size. In our experiments, we tested a range of bundling parameters, with the goal of producing bundlings with a large average bundle size. Section 3 will show how well we were able to counteract the effects of network latency.

3 Experimental results

This section describes experiments we performed to evaluate the effectiveness of bundling compared to other network class loading mechanisms.

There are two fundamental ways to apply bundling. First, it may be applied to the class and resource files used by a single application, based on profiles collected from that application. This approach is reasonable if the intent is to optimize the transfer of files used by a single application in isolation. The second way it can be applied is to a library used by many applications. Applying bundling to a library used by multiple applications is a more difficult problem than applying it to a single application, because the class loading behavior of different applications is much less consistent than the class loading behavior of multiple runs of a single application. (Note that there is no absolutely rigid distinction between these approaches. For example, an office application might have a common library of classes as well as several 'sub-applications', such as word processor, spreadsheet, etc. The loading behavior for the classes and resources within the sub-applications would likely be quite consistent from run to run. However, each sub-application might use the common library in different ways, resulting in inconsistent loading behavior within the classes and resources of the library from run to run.)

In order to test both extremes (multiple applications *vs.* single application), we generated bundlings from a subset of JDK 1.2.2's `rt.jar`, which contains the standard Java libraries used by all applications. The subset includes AWT, Swing, and Java2D, but does not include the 'core' packages (`java.lang.*`, `java.util.*`, etc.) Since almost all interactive Java applications use AWT, Swing, or Java2D, we felt the `rt.jar` subset was a good example of a substantial library used by many applications in nontrivial ways.

The first experiment uses profiles from several applications and applets to generate bundlings for the subset of `rt.jar`. Since the applications load different class files and resources in significantly different orders, this experiment represents a 'stress test' of the bundling algorithm. We would like to emphasize that applying bundling to a large, multi-purpose library is not an ideal application of bundling, since bundling is designed to take advantage of regularities in class loading behavior.

The second experiment uses profiles from a single application to generate bundlings for the `rt.jar` subset. Since there is more regularity in the class and resource loading behavior of a single application than in several different applications, this experiment represents an ideal scenario for bundling, in which we would expect its performance to come close to that of cumulative compression in terms of the compression ratio achieved, while avoiding the high latency delays associated with on-demand loading.

The third experiment measures the effectiveness of the bundlings generated in the first experiment on two applications not represented in the input profiles used to generate those bundlings. This experiment is an even less ideal application for bundling than the first experiment; however, it does offer some insight into how the performance of bundling may degrade when it encounters class loading behavior not represented in the input profiles.

The fourth experiment measures the startup time for one of the test applications using simulated network bandwidth and latency. The experiment is designed to determine how well bundling performs in a real JVM, as opposed to off-line simulations.

3.1 Bundling parameters

In generating the bundlings for the experiments, we tried three combinations of bundling parameters:

- Minimum edge weight 1.0, maximum bundle size 200, maximum bundle spread 5. These bundling parameters are relatively strict, in the sense that they forbid two files from being placed in the same bundle if there were any profiles in which one was loaded without the other. They also significantly limit the extent to which files can be delivered ahead of when they are needed. These parameters produced a bundling where the average bundle size was quite small, on the order of 3 or 4 files.
- Minimum edge weight 0.8, maximum bundle size 1000, maximum bundle spread 200. This is a 'looser' set of parameters, which permits files to be placed in the same bundle even if they aren't always loaded together, and also allows files to be sent significantly ahead of when they are needed. These parameters produced a bundling where the bundles were larger, the average bundle size being about 12 files.
- Minimum edge weight 0.8, maximum bundle size 1000, maximum bundle spread 500. This is an even looser set of parameters, which produced bundles whose average size was about 24 files.

Note that in all cases, the maximum bundle size parameter was not reached, so bundle growth was only constrained by the minimum edge weight and maximum bundle spread.

In the figures and text, the parameters used to generate a bundling are specified in the form w-m-s, where w is the minimum edge weight, m is the maximum bundle size, and s is the maximum bundle spread.

3.2 Network parameters

We tried two combinations of network bandwidth and latency in our simulations. First, we tried 500,000 bytes/second bandwidth and 4 ms latency, which is typical of what one might see on a wireless local area network. Second, we tried 50,000 bytes/second bandwidth and 70 ms latency, which is typical of a high-speed wide area network.

3.3 Measurements

Three types of data were collected.

First, for experiments 1–3, we performed a simulation of expected arrival times for each file loaded in the test profiles, based on network bandwidth and latency. The idea is to assume that the client issues requests for each file in the profile one at a time, and that the requested file must arrive before the next file in the profile can be requested. A request for a file which has not arrived yet must be sent to the server. Only one bundle may be in transit at any given moment, and the data for a requested bundle starts arriving only after the latency period has elapsed following the request, or after all bundles in transit have finished transferring, whichever is later. We take into account the relative offsets of each file within a bundle, and their sizes within the bundle. While this model is somewhat simplistic, it does take into account the most important factors in network class loading, since the computational overhead of network class loading is generally less significant than the network overhead. The expected arrival time for each file in the test profile is compared with the 'ideal' arrival time, which is simply the time the file would arrive if a single bundle containing all of the files in the profile in the correct order were downloaded. (This is like on-the-fly compression, except that the entire sequence of requests is 'known' in advance.)

Second, for the applications in experiment 1, we measured the total number of bytes transferred from server to client. The fewer bytes transferred, the higher the compression ratio and the less bandwidth consumed. Note that even though our current implementation supports only bundles in zlib format, we were able to accurately measure the download sizes for Pack bundles based on the bundle transfer behavior observed for zlib bundles.

Third, in experiment 4 we measured the startup time for one of the test applications (Argo/UML) under simulated network bandwidth and latency conditions. By running an actual JVM under realistic conditions, we are able to see how successfully our estimates of class loading performance (the expected arrival times described above) predict real application performance. In particular, this experiment takes into account other sources of overhead in the JVM, such as loading native libraries, verifying class files, etc.

3.4 First experiment

In the first experiment, bundlings of the `rt.jar` subset were generated using a collection of 17 input profiles, from the following applets and applications:

- Sun's Java2D demo
- Argo/UML, an object-oriented design tool
- Sun's Swing demo
- The JDK 1.2.2 demo applets
- The Jazz HiNote zoomable user interface demo (from the University of Maryland's Human-Computer Interaction Lab)
- The jEdit editor, version 2.3pre2

We evaluated the bundlings produced from the input profiles on five applications: the Drawtest applet, the TicTacToe applet, Argo/UML, the Java2D demos, and HiNote. For all five applications, the profile used in the experiment was a member of the input profiles used to generate the bundlings.

Figures 1–5 show the expected file arrival times vs. ideal file arrival times for the applications. In general, the bundlings were competitive with the ideal arrival times, although the 'strict' bundling 1.0-200-5 fared noticably worse than the 'loose' bundlings. This shows that even when network latency is relatively low (4 ms) it is still an important factor.

The vertical 'cliffs' in the plots correspond to files being sent earlier than when they are needed; essentially, they delay the file whose request is currently outstanding. The diagonal sections (where the slope diverges from the ideal) corresponds to request latency. The horizontal 'plateaus' in the plots correspond to files already received being used by the client. These satisfy their requests instantaneously, since the data is already available.

Figure 6 shows the download sizes for the applications and bundlings in experiment 1, for bundles in both zlib and Pack formats. All of the zlib bundlings were competitive with cumulative zip, although they were much larger than an equivalent Pack archive would have been. The 'loose' Pack bundlings (0.8-1000-200 and 0.8-1000-500) were able to get quite close to the compression ratio of a single Pack archive, while the 'strict' Pack bundling (1.0-200-5) had somewhat lower compression than a single Pack archive. All of the Pack bundlings were significantly better than cumulative zip.

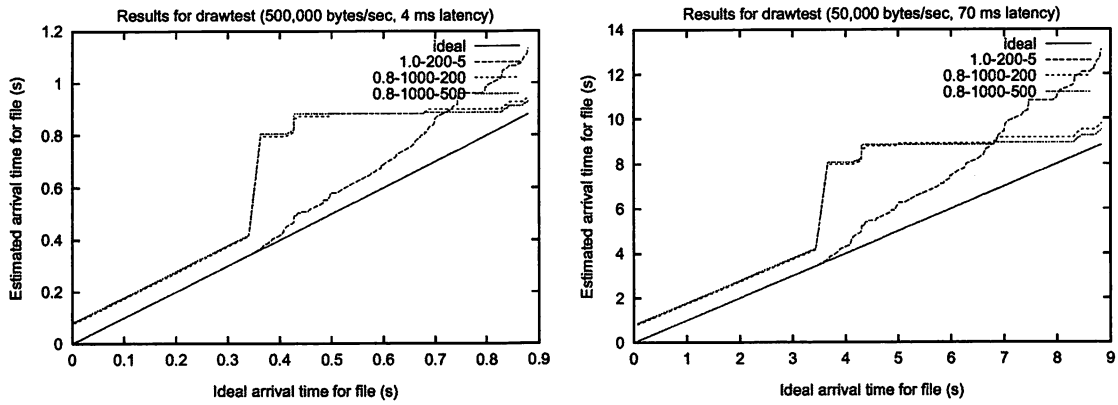


Figure 1: Ideal file arrival times vs. estimated file arrival times for drawtest (experiment 1).

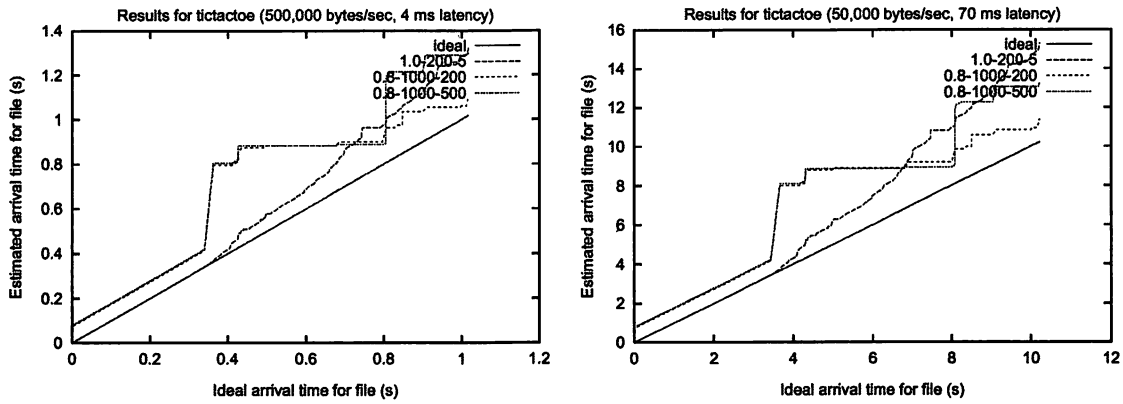


Figure 2: Ideal file arrival times vs. estimated file arrival times for tictactoe (experiment 1).

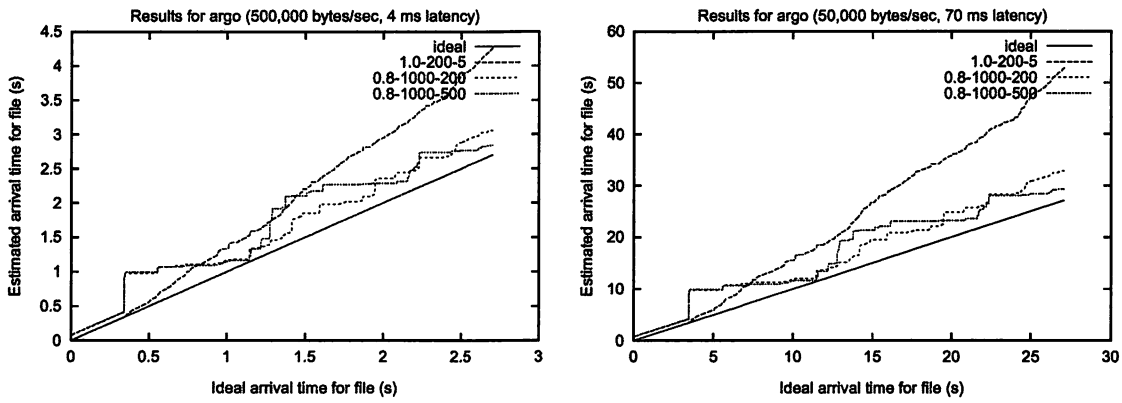


Figure 3: Ideal file arrival times vs. estimated file arrival times for Argo/UML (experiment 1).

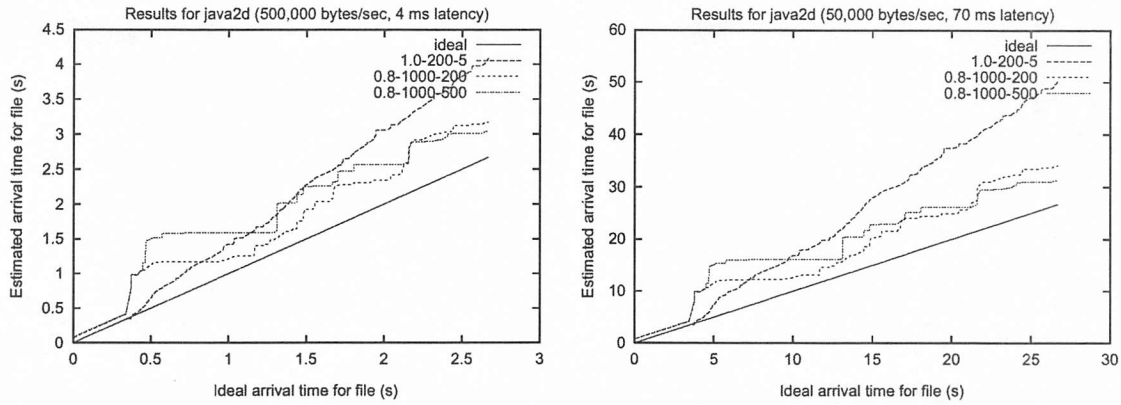


Figure 4: Ideal file arrival times vs. estimated file arrival times for Java2D (experiment 1).

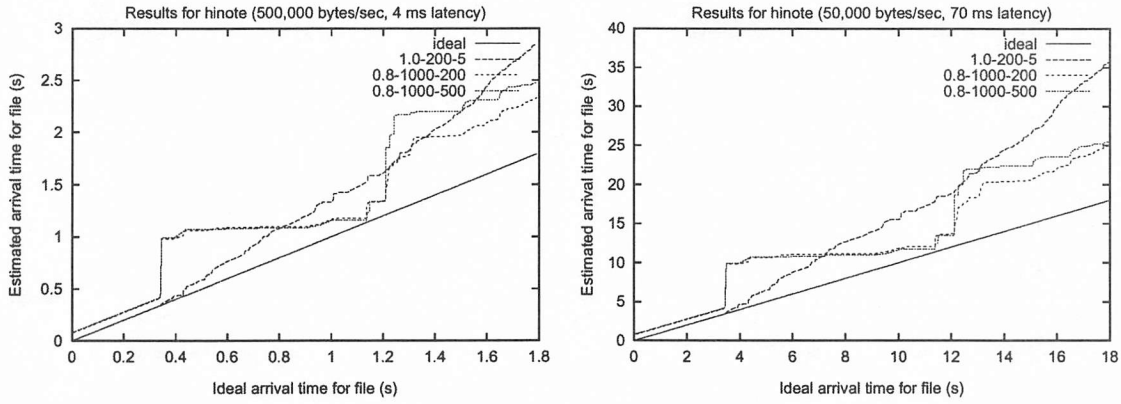


Figure 5: Ideal file arrival times vs. estimated file arrival times for HiNote (experiment 1).

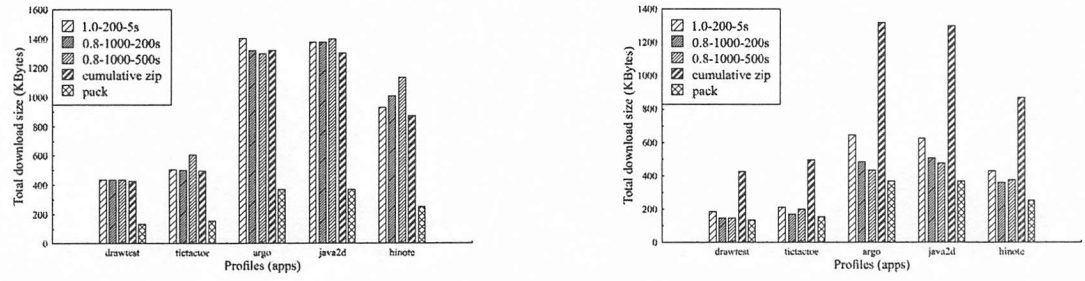


Figure 6: Download sizes for zlib bundles and Pack bundles vs. cumulative zip and Pack (experiment 1).

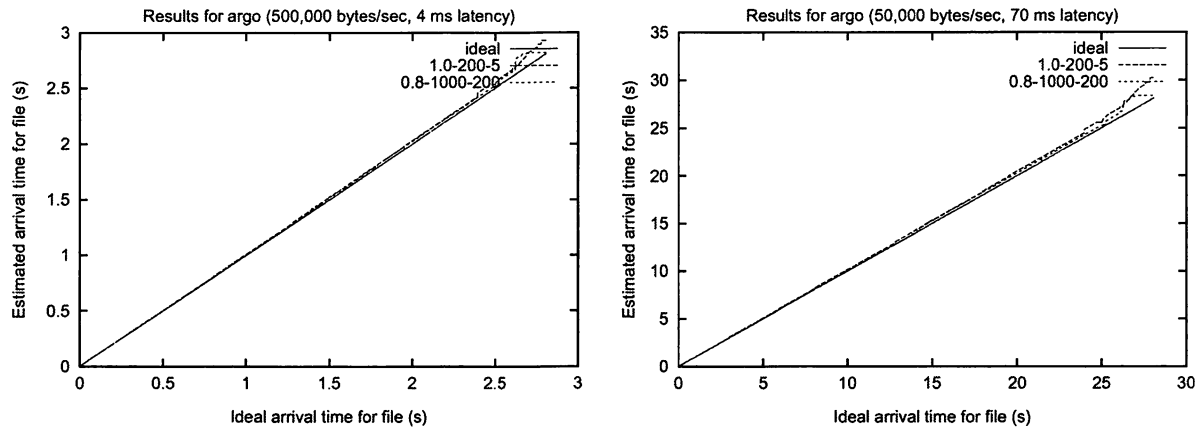


Figure 7: Ideal file arrival times vs. estimated file arrival times for Argo/UML (experiment 2).

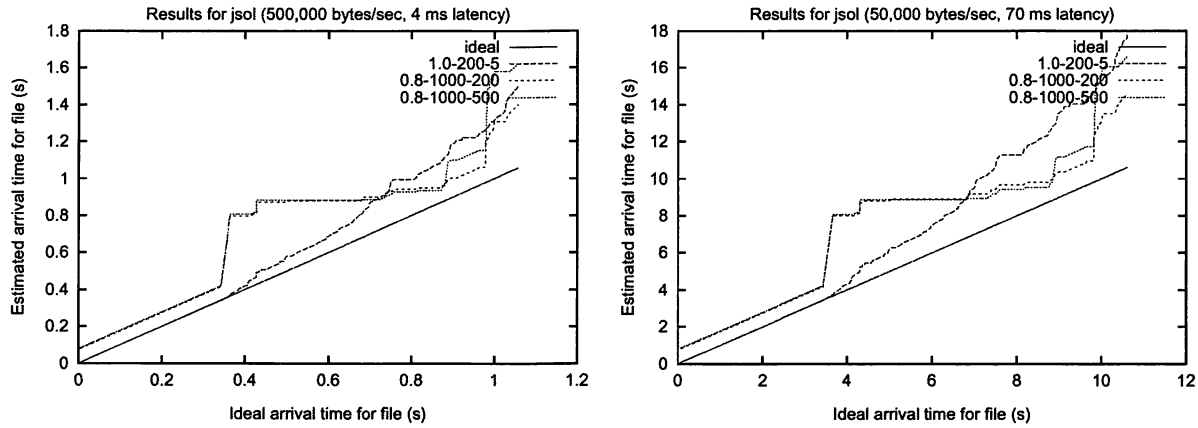


Figure 8: Ideal file arrival times vs. estimated file arrival times for jsolitaire (experiment 3).

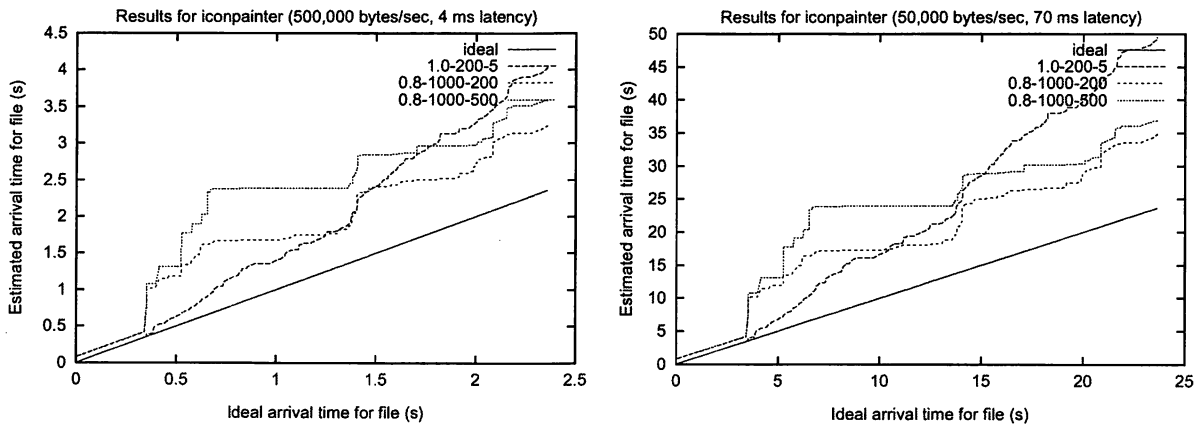


Figure 9: Ideal file arrival times vs. estimated file arrival times for iconpainter (experiment 3).

Minimum edge weight	Maximum bundle spread	Number of bundles	Startup time (seconds) 500,000 bytes/s 4 ms latency	Startup time (seconds) 50,000 bytes/s 70 ms latency	Number of unused files transferred
n/a	n/a	1	22.47	44.74	0
1.0	5	317	24.54	67.77	0
0.8	200	88	22.57	48.85	57
0.8	500	30	22.50	46.46	99

Table 2: Startup time for Argo/UML under simulated network bandwidth and latency.

3.5 Second experiment

In the second experiment, bundlings of the `rt.jar` subset were generated from 5 profiles collected from a single application, Argo/UML. Each profile exercised different application functionality. Then, arrival time data were collected for a single profile (which was a member of the profiles used to generate the bundlings).

Figure 7 shows the expected file arrival times vs. ideal file arrival times. As expected, the bundlings achieve file arrival times very close to the ideal file arrival times. Because of the regularity in class loading behavior in different runs of the same application, only a few bundles were generated, increasing the bundle size and compression ratio, and decreasing the cost associated with making requests to the server.

(Note that bundling 0.8-1000-500 is not shown because it is identical to 0.8-1000-200 for the profiles used.)

3.6 Third experiment

In the third experiment, we measured the effectiveness of the bundlings of the `rt.jar` subset produced in experiment 1 on two applications not represented in the input profiles used to generate those bundlings. The applications were jsolitaire (a Solitaire card game applet) and iconpainter (an icon editor). Note that the iconpainter used four class files not loaded in any of the original input profiles; we added these to the bundlings as bundles of size 1 (i.e., containing a single class file).

Figures 8 and 9 show the expected vs. ideal file arrival times for jsolitaire and iconpainter, respectively. The arrival times for the bundlings are somewhat further from ideal than in experiment 1, where the profiles tested were members of the set of profiles used to generate the bundlings. This shows that bundling works best for applications which are represented in the input profiles used to generate the bundlings. However, the time for all of the files to arrive for the loose bundlings is only about 40% later than for the ideal case. Considering that bundling typically achieves twice the compression of Jar archives, this overhead may be acceptable in some situations.

3.7 Fourth experiment

In the fourth experiment, we measured the amount of time needed by the Argo/UML application to initialize and display its user interface, using the bundle class loader client and server for the `rt.jar` subset. We simulated network bandwidth by restricting the rate at which the server sent data, and we simulated network latency by instrumenting the server to pause for a fixed amount of time before processing a client request. The client and server communicated over local TCP/IP on a two-processor Sun Ultra 60 workstation.

Table 2 shows the results. The first row of the table gives the startup times for a single 'ideal' bundle consisting of all of the required files in the correct order. The other rows show the startup times for the bundlings used in experiment 1. For the high bandwidth and low latency case, the startup times for the bundlings were almost identical to the ideal startup time. For the lower bandwidth, higher latency case, the 'loose' bundlings performed much better than the 'strict' bundling, even though some unneeded files were transferred.

4 Related work

Pack [9] is a compressed archive format for Java class files developed by William Pugh. While it achieves very good compression, it has a relatively slow decompressor, so is most useful for slow networks.

Jazz [3], developed by Quetzalcoatl Bradley, R. Nigel Horspool, and Jan Vitek, is another compressed archive format for Java class files. It is similar to Pack, but uses a somewhat less aggressive compression scheme.

In [11], Tip, Laffra, Sweeney, and Streeter describe Jax, an application extractor for Java. Jax uses a number of whole-program analysis techniques to transform an application's class files to reduce its size. For example, Jax detects and eliminates unused methods and fields. One motivation for size reduction is reducing download time when loading the application from a network. The techniques employed by Jax are largely orthogonal to compression and bundling, and would be complementary to

the techniques we present in this paper.

In [6], Krintz, Calder, and Hölzle describe techniques for reducing transfer delay by splitting classes (methods and data members) into hot and cold parts, and for prefetching classes. The class file splitting technique transforms classes to avoid downloading code for rarely used methods and fields. The prefetching technique tries to avoid request latency and transfer delay by placing requests for class files prior to the 'first-use' points for those classes. Because these techniques rely on the client issuing an explicit request for each file, they would only by suitable for use with individual file compression or on-the-fly compression.

Appwerx Expresso [2] is a commercial product that reduces applet startup time by reordering class file and resources in a Jar file and implementing a class loader that can concurrently download and execute classes from the Jar file.

In [7], Kuenning and Popek describe a technique for hoarding files on a mobile computer prior to disconnection from a network. Although the problem they address is different, their approach is similar to ours in that they use profiles of file access by the user to group related files together and to predict which files will be needed by the user. However, there are substantial differences between their techniques and ours. Their notion of 'semantic distance' between two files is based on the number of intervening file accesses to other files in the profiles, and on the relative orders of open and close operations on files, while our notion of edge weight is based on the frequency of the files being loaded in the same profile. (Because their profiles are system-wide, they do not have the notion of profiles specific to an application.) Whereas our motivation for grouping files into bundles is to enable better compression and for pre-sending to reduce request latency, their motivation is to detect related groups of files in order to ensure that entire 'projects' are hoarded, rather than just recently used files in isolation.

5 Conclusions

We described a technique called *bundling* which splits a collection of class files and resources into bundles based on previously observed class and resource loading behavior. We showed that bundling is competitive with cumulative compression when the applications and profiles are known in advance, and that it is no worse than the Jar archive format when used on an application not included in the training set.

One possibility for future work is to use a static dictionary of commonly occurring class file data in compressing the bundles. This could improve the compression ratio on small bundles.

Another possibility for future work is to apply bundling in other contexts where the transfer of sequences of files is required. For example, it might be applicable in serving a collection of files to be partially mirrored at other sites.

Acknowledgements

We would like to thank Jeremy Manson for his helpful comments and suggestions. We would also like to thank the anonymous reviewers for identifying weaknesses in the original version of the paper.

Notes

[1]Java is a trademark of Sun Microsystems.

[2]Note that for on-the-fly compression to work, the compression algorithm must have the ability to flush the compressed stream at arbitrary points (i.e., the file boundaries). Zlib has this ability, but it is not implemented in the `java.util.zip` package.

[3]For HTTP 1.1 persistent connections.

[4]Not implemented in Java HTTP client.

[5]Jar indexes can support an arbitrary request granularity.

References

[1] zlib Home Site. http://www.info-zip.org/pub/infozip/zlib/.

[2] Appwerx, Inc. Appwerx Expresso. http://www.appwerx.com/expresso/.

[3] Quetzalcoatl Bradley, R. Nigel Horspool, and Jan Vitek. JAZZ: An Efficient Compressed Format for Java Archive Files. In *Proceedings of CASCON'98, Toronto, Ontario*, 1998.

[4] L. P. Deutsch. RFC 1951: DEFLATE compressed data format specification version 1.3, May 1996.

[5] L. P. Deutsch and J-L. Gailly. RFC 1950: ZLIB compressed data format specification version 3.3, May 1996.

[6] Chandra Krintz, Brad Calder, and Urs Hölzle. Reducing Transfer Delay Using Java Class File Splitting and Prefetching. In *Proceedings of the 1999*

ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, Denver, Colorado, 1999.

[7] Geoffrey H. Kuenning and Gerald J. Popek. Automated Hoarding for Mobile Computers. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[8] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications, Vancouver, Canada*, 1998.

[9] William Pugh. Compressing Java Class Files. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation, Atlanta, Georgia*, 1999.

[10] Sun Microsystems. JDK 1.3 — JAR File Specification, 1999.

[11] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical Experience with an Application Extractor for Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, Denver, Colorado*, 1999.

Deterministic Execution of Java's Primitive Bytecode Operations

Fridtjof Siebert
IPD, Universität Karlsruhe
Am Fasanengarten 5
76128 Karlsruhe, Germany
siebert@ira.uka.de

Andy Walter
Forschungszentrum Informatik (FZI)
Haid- und Neu Straße 10-14
76131 Karlsruhe, Germany
anwalt@ira.uka.de

This work was partially funded by the DFG program GRK 209

ABSTRACT

For the application of Java in realtime and safety critical domains, an analysis of the worst-case execution times of primitive Java operations is necessary. All primitive operations must either execute in constant time or have a reasonable upper bound for their execution time. The difficulties that arise for a Java virtual machine and a Java compiler in this context will be presented here. This includes the implementation of Java's class and interface model, class initialization, monitors and automatic memory management. A new Java virtual machine and compiler that solves these difficulties has been implemented and its performance has been analysed.

1. INTRODUCTION

Java [AG98] is becoming increasingly popular for software development, even in application domains well beyond the original target domain of the language. The driving force is the high productivity, code quality, security and platform-independence that typically goes along with the use of Java. Java is promoted even as a development tool for realtime critical systems that require deterministic execution [JCons00, RTJEG00], even though there are several obstacles to be overcome to achieve deterministic execution of Java code.

The most obvious source for indeterministic execution is the use of garbage collection in Java. Realtime embedded systems such as in industrial automation, avionics or automotive applications often require short response times. Blocking garbage collectors cause long pauses that are hard to predict and unacceptable for these applications.

Although incremental garbage collection techniques can help to reduce the likelihood for a blocking garbage collection pause, they can not guarantee it. It can still occur that the collector does not make sufficient progress and does not catch up with the application. Consequently, the system can fail or require long blocking pauses to recycle memory or defragment the heap.

A deterministic implementation of Java must provide means to determine worst-case execution times for Java's primitive operations. The dynamic structure of Java, with inheritance, virtual method calls and multiple-inheritance for interfaces, poses several difficulties for the implementation. The time required for calls or type checks must be limited and statically determinable.

Further difficulties are caused by the dynamic nature of Java's class loading and initialization mechanism. Finally, Java's synchronization primitives that permit to have a monitor associated with any Java object and Java's exception mechanism pose further difficulties for a deterministic implementation.

2. RELATED WORK

Research on object-oriented language implementations so far focused on high average-case runtime performance. A predictable runtime cost, that is required in realtime systems, is of little importance for classical Java applications.

For the use of Java in realtime applications, Nilsen identifies the analysis of conservative worst-case execution times for code sequences that exclude recursion and unbounded loops as a key enabling technology [Nilsen96, NR95]. This analysis is not straightforward for all of Java's primitive operations; this

paper will describe how a Java implementation can permit this analysis. The other enabling technologies listed by Nilsen are execution time measurement, rate-monotonic analysis [LL73], static cyclic schedules and realtime garbage collection.

A number of publications describe technologies to efficiently implement object-oriented languages or Java specific features like monitors, improving the average-case runtime and memory costs.

Different approaches for the implementation of dynamic dispatching in the context of single- and multiple-inheritance have been proposed. Zendra et. al. suggest using specific dispatch functions instead of virtual function tables to make better use of modern processor architectures [ZCC97]. Typical implementations of *C++* [Strou87] use several virtual function tables per object and direct references to sub-objects to implement multiple inheritance. To improve the average performance of virtual calls, inline caching has been suggested [DS84].

Yang et. al. [Yang99] and Bacon et. al. [BKMS98] presented techniques to inline monitors in objects and reduce the average runtime cost for monitor operations to a minimum. However, their approaches cannot avoid high worst-case overhead in the case of contention.

3. THE JAMAICA VIRTUAL MACHINE IMPLEMENTATION

Jamaica is a new virtual machine implementation for Java and a static Java bytecode compiler. It provides deterministic execution of the whole Java language, including deterministic garbage collection, and constant time execution of the primitive Java operations.

Jamaica uses a builder utility to create stand-alone applications out of a set of Java class files and the Jamaica virtual machine. The builder can perform smart linking and compilation of the Java application into optimized *C* code using a static Java bytecode compiler. The compiler implements several optimizations similar to those described by Weiss et. al. [Weiss98]. The generated *C* code is then translated into machine code by a C-compiler such as *gcc*.

To reduce the footprint of Java applications, interpreted compact byte code and compiled machine code can be mixed. Profiling data can be used to guide the compiler to compile only the hot spots of the applications and achieve the best speedup with a small increase in code size.

3.1 Executing Bytecodes

Most Java bytecode instructions can be implemented directly as a short sequence of machine instructions that executes in constant time when cache effects are ignored. Even in the presence of processor caches, a short worst-case execution time can be determined easily. These operations include accesses to local variables and the Java stack, arithmetic instructions, comparisons and branches.

The bytecode instructions for which a deterministic implementation is not straightforward will be described in detail in this section.

3.1.1 Loading String Constants

One needs to be careful when loading string constants with the *ldc* bytecode instruction. If this instruction is used to load a string object, this string object must have been created in advance to avoid unpredictable allocation overhead. The implementation achieves this by creating all constant string objects at class load time. During execution, loading the string's address is all that needs to be done.

3.1.2 Class Initialization

The semantics of Java require that on the first access to a static field, a static method or the first creation of an instance of a class the corresponding class be resolved and its static initializer be executed. In a standard Java implementation, the first resolution of a class also causes the class to be loaded, causing a very complex operation that might take much time. The resolution causes very long worst-case execution times of the primitive operations that might cause class resolution, while the actual execution time of these operations is typically very short once the referred class is initialized.

This problem is not easy to solve without changing Java's semantics. The Java virtual machine specification [LY99] explicitly allows early loading and resolution of classes, as long as the semantics of Java are respected. However, this does not allow early execution of static initializers, since this would change the control flow and hence the semantics of the implementation.

To avoid the overhead of loading and linking a class during class resolution, the Jamaica virtual machine recursively loads and links all classes that are referenced by the root class at system initialization. If later during execution of the system additional classes are loaded using methods like *java.lang.Class.forName()* or the reflection API, this process is repeated and all referenced classes are loaded as well.

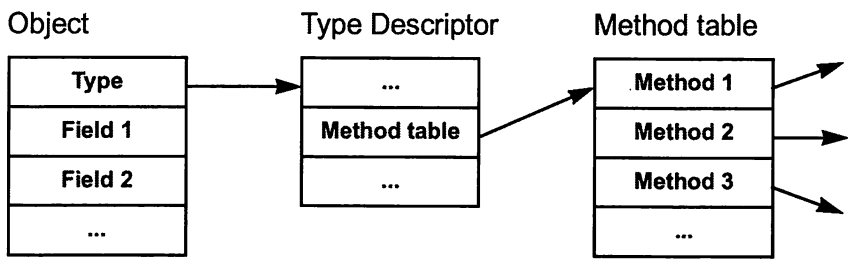


Figure 1: Method table used for dynamic binding of virtual calls

What is left at the first reference to a class is the execution of its static initializer. The user is free to perform arbitrarily complex calculations within the static initializer, so a worst-case execution time for this operation can not be guaranteed by the Java implementation unless the static initializer is relatively simple, e.g., code that contains no method calls or loops.

Even for simple static initializers, the call overhead causes fairly bad worst-case execution times for simple operations like an access to a static field. The user can avoid this overhead by explicitly causing early initialization of the referred class during system startup.

The resulting overhead for operations that cause class resolution is then reduced to the overhead of checking the initialization state of the class. This can be done by reading a single field in the class' descriptor and a conditional branch.

We plan to improve the Jamaica compiler such that explicit early initialization can be detected statically such that the test can be avoided in most cases.

3.1.3 Method Invocation

The invocation of methods in the context of class extension and interface implementation is another difficulty for a deterministic implementation of Java. There are four different bytecode instructions for the invocation of methods.

Static Calls

Two call instruction avoid dynamic binding altogether: *invoke_static* and *invoke_special*. The first one is used to call static methods, while *invoke_special* is used for instance methods, but uses static binding instead of dynamic binding (this is used for object initialization or when explicit calls to a certain class' method is needed as in *super.method()*). The lack of dynamic binding semantics in these two bytecode instructions makes the implementation straightforward. Control flow can directly switch to the called method since the target method is known.

Virtual Calls

A call using *invoke_virtual* is slightly more difficult to implement since the dynamic type of the target object needs to be taken into account. If the called method was redefined by the dynamic type of the target, that redefined method needs to be called.

The standard method of implementation provides deterministic execution time. A virtual call is implemented with a method table that is part of the type descriptor of each object (**Figure 1**). Each method is assigned a unique index in the method table. Inherited methods keep their unique indices in the new class. Methods defined within a class are assigned the next available index, while methods that redefine inherited methods inherit the index of the original method. Every method is recorded in the method table at its index, while inherited methods that were redefined are replaced by their redefined versions.

This technique permits one to find the target method of a virtual call very efficiently. All that is required is a lookup in the method table at the method's index. If the method has been redefined, the redefined method's entry will use the same index as the original method and the redefined method will be found. Using this technique, a virtual call can be performed in constant time.

Interface Calls

The most difficult to implement calls are calls to interface methods via the bytecode operation *invoke_interface*. The reason is that multiple inheritance

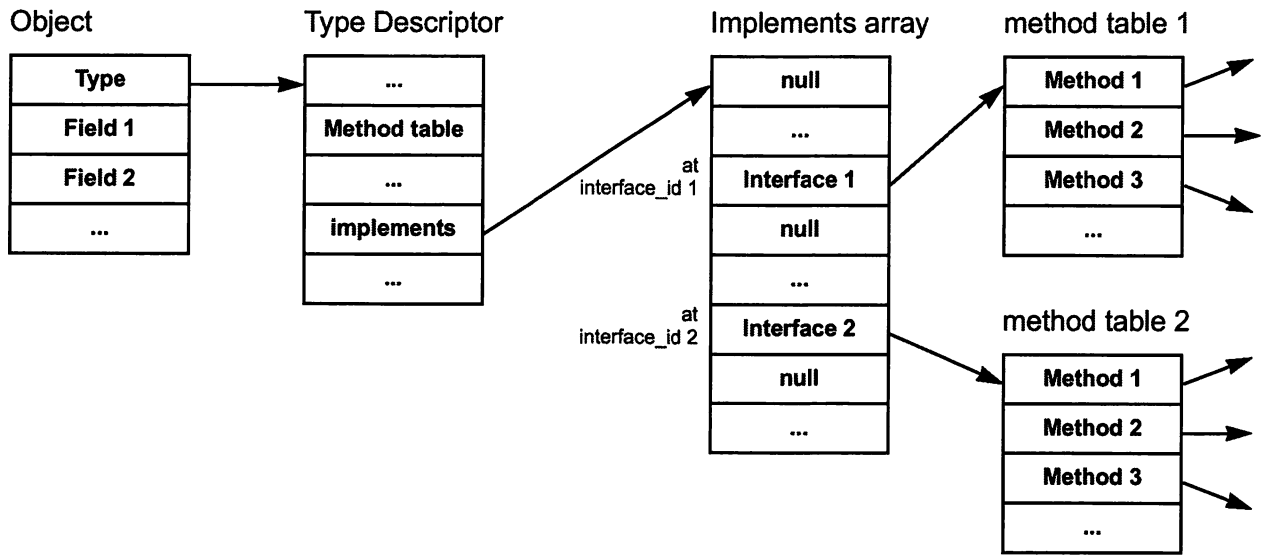


Figure 2: Interface call using implements array

makes it impossible to assign constant indices to the methods as in the case of virtual calls. A dynamic search for the called method is typically made in this case. This causes a call overhead that depends on the number of methods in a class or the number of interfaces implemented by a class. Therefore calls to interface methods are significantly less efficient in many Java implementations and the execution time depends on the structure of the target class.

The representation of classes and interfaces that was chosen for the Jamaica implementation permits constant-time calls to interface methods. Every interface is assigned a unique identifier *interface_id*, starting at *0* for the first interface loaded. Each method within an interface is assigned an *interface_method_index* similar to the methods of normal classes. The class descriptor of every class that implements one or several interfaces contains a reference to an array of references. This array is referred to as *implements* array. For each interface that is implemented, the *implements* array contains a valid reference at the interface' *interface_id*. This reference points to an array of the methods defined in the interface and implemented in the class, with each method at its *interface_method_index* (**Figure 2**). All entries in the *implements* array that correspond to interfaces that are not implemented by the class are set to *null*. For the call of an interface method, all that is needed is to read the *implements* array from the target object's type descriptor, index this array at the interface' *interface_id* and finally read the method at the *interface_method_index*. Compared to virtual methods, there is only one additional indirection needed, calls to interface methods are performed in constant time.

Nevertheless, there is an important disadvantage of this approach: the size of the *implements* array is linear in the number i of interfaces, and the total memory required for a system with c classes is therefore in $O(c \cdot i)$. The memory usage is quadratic in the size of the code. With the applications executed using Jamaica, this has not caused any difficulties, but it might be a problem for very large applications.

The memory overhead can be reduced using some simple optimizations. It is sufficient if the size of the *implements* array is reduced to *interface_id*+1 for the largest *interface_id* of all the interfaces implemented. In addition, all classes that do not implement any interfaces can share one single empty *implements* array. Jamaica makes use of these optimizations.

Dynamic loading of interface classes can also be handled using this mechanism. New interface classes will be assigned new *interface_ids* and classes implementing this interfaces will get sufficiently large *implements* arrays. In the context of class garbage collection, care must be taken to ensure that unused *interface_ids* will be reused, so that the size of the *implements* arrays does not grow without limit.

One can imagine that the typically high additional overhead for calls due to multiple inheritance might be the only reason for the existence of interfaces.

Using the scheme presented here, the overhead is minimal compared to a virtual call using single inheritance. If Java allowed multiple inheritance for normal classes, there would be no need for interfaces at all. A single concept, the class, could be used to model interfaces and classes. This would remove the often difficult decision whether to model a concept using an abstract class or an interface. The former is more powerful by allowing the implementation of some methods, while the latter is more flexible and allows multiple inheritance. Object-oriented programming languages like Eiffel have shown that a single concept using multiple inheritance is sufficient and that efficient implementation of multiple inheritance is possible [Meyer92].

Additional Call and Return Overhead

Apart from the overhead needed to determine which method is called, a call requires setting up a new stack frame. Later, this stack frame has to be removed when the method returns. Unlike in languages like *C* or *C++*, the creation and removal of the stack frame is not a constant time operation. The reason is that reference values on the stack frame must be known to the garbage collector. Each entry in the stack frame that is reserved for a memory reference must be at least initialized at the start of a method call and also perhaps cleaned up at the end of the call. Consequently, the overhead for creation or removal of a stack frame is linear in its size.

3.1.4 Type Checking

Another problem area for a deterministic implementation that is closely related to dynamic calls is dynamic type checking. There are two situations in Java code that require dynamic determination of the type of a reference: explicit type tests using the *instanceof* operator and type casts. Consequently, there are two bytecode operations, *instanceof* and *checkcast*, for these two purposes. The first of these, *instanceof*, tests if a referenced object is of a certain type and produces a *boolean* result, while *checkcast* causes an exception if the type test fails, and does nothing otherwise. Furthermore, there are three categories of types in Java that have to be treated differently in the type check: classes, interfaces, and arrays.

Type Checking for Classes

The most common case for a type check is checking whether or not a referenced object is an instance of a certain class. As an example, assume the following *if*-statement.

```
if (r instanceof A) {
  System.out.
    print("r is of class A");
}
```

The straightforward implementation of this type check would traverse the inheritance-chain of the object referenced by *r* until either class *A* is found or the root object *java.lang.Object* has been reached. This implementation requires time linear in the depth of the inheritance tree, a worst-case execution time for the type check is difficult to determine.

A simple modification of the representation of inherited classes permits constant-time type checking for classes. Every class has a fixed position in the inheritance tree and a fixed distance to class *java.lang.Object*. All classes in the inheritance tree that are on the path from a class *C* to *java.lang.Object* are referred to as *C*'s ancestors. *C*'s ancestors include the classes *C* and *java.lang.Object*. The number of ancestors of class *C* is the *depth* of *C* in the inheritance tree. Any class descriptor can now be equipped with a reference to an array of all ancestors, as shown in **Figure 3**. Each ancestor that resides at position *depth* in the inheritance tree is stored at position *depth-1* in this *ancestors* array. Now, the type check can be done in constant time. All that is needed is a check of the *ancestors* array's entry at the position corresponding to the class' depth.

C-code that performs the *instanceof*-check shown above would look like this.

```
if ((r!=null                              )
 && (r->type->
        ancestors.length>=A->depth)
 && (r->type->
        ancestors[A->depth-1] == A))
{
  System.out.
    print("r is of class A");
}
```

The code requires reading the referenced object's class descriptor, the *ancestors* array, the array length and one element in the array. The *depth* of class *A* and the class descriptor of class *A* are also required in this test, but these values are runtime constants and can be inlined by a compiler.

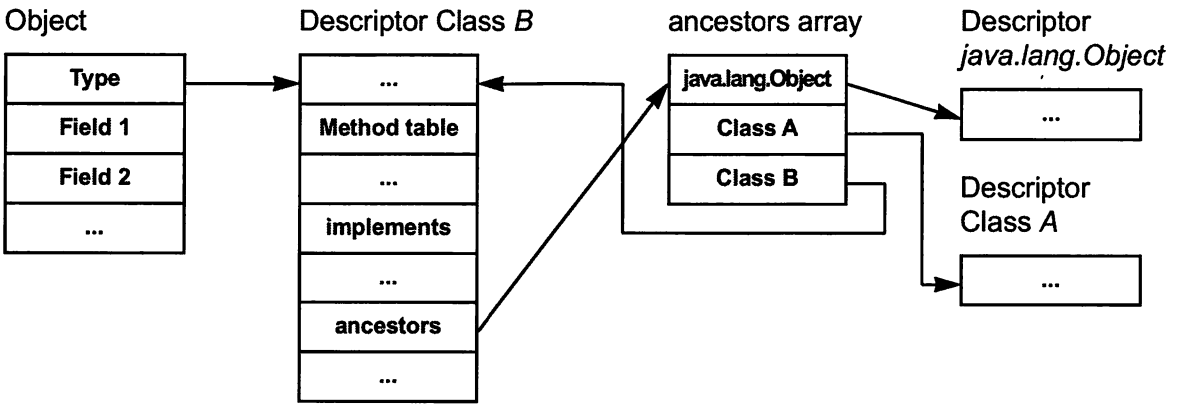


Figure 3: Type checking for classes using ancestors array. Class *B* extends class *A* which inherits directly from class *java.lang.Object.*

Type Checking for Interfaces

The *ancestors* array approach used for class types cannot be used directly for interfaces since a class might implement several interfaces that reside at the same *depth* in the inheritance tree. What is needed is an array with a unique entry for every interface that is implemented by a class. Fortunately, such an array was already introduced for the efficient implementation of interface calls: the *implements* array.

For a type check of the form

```
if (r instanceof I) {
  System.out.
    print("r is of interface I");
}
```

all that is needed is to check whether the *implements* array of *r* has an entry for interface *I*. In C-like code, this might look as follows.

```
if ((r!=null                          )
 && (r->type->
        implements.length >  I->id)
 && (r->type->
        implements[I->id] != null ))
{
  System.out.
    print("r is of interface I");
}
```

The code requires reading the class descriptor reference, the implements array, its length and the entry corresponding to the interface *I*. The *interface_id* that is required is a runtime constant.

Type Checking for Arrays

The type checking semantics for arrays are defined recursively. The examined type must be an array. If this is the case, the type checking continues for the element type of the array. A straightforward implementation of this definition requires execution time linear in the number of dimensions of the array. Nevertheless, a constant-time implementation of type checking for multidimensional arrays is possible. One need only store the dimension count and a reference to the final non-array element type with the array type.

This can be illustrated using the following code sequence.

```
if (r instanceof A[][][]) {
  System.out.
    print("r is of type A[][][]");
}
```

All that is required for this test is to check whether the object referred to by *r* is an array of dimension three and the base element type *A*. The C-code might look like this:

```
if ((r!=null                              )
 && (r->type->dimensions == 3             )
 && (r->type->elemnt->
        ancestors.length >=A->depth)
 && (r->type->elemnt->
        ancestors[A->depth-1] == A))
{
  System.out.
    print("r is of type A[][][]");
}
```

The code checks the correct number of dimensions of the array and then does a type check for the element class as described above.

Care needs to be taken when checking arrays of class *java.lang.Object* or interfaces *java.lang.Clo-*

neable and *java.io.Serializable.* The class *java.lang. Object* is an ancestor of all arrays and the two interfaces are implemented by all arrays. To implement the type check for these arrays correctly, the check must additionally succeed whenever the dimension of the examined type is higher than that with which the array type it is compared.

3.1.5 Switch Statement

The Java bytecode provides two different operations for *switch*-statements: *tableswitch* and *lookupswitch*. For a *switch*-statement with *case*-entries that are similar values, *tableswitch* is used. It permits a constant-time table lookup for the statement sequence that needs to be executed.

For *case*-entries that extend over a large range of values with unused intervals *tableswitch* would require a large table. In this case, *lookupswitch* is used in the bytecode. This instruction uses a sorted table of *case*-value and target address pairs. The lookup requires a binary search of this table, hence the runtime is logarithmic in the number of *case*-labels in the switch statement.

The Java implementation of the interpreter has little choice here but to implement the binary search, converting a *lookupswitch* into a constant-time *tableswitch* might require too much memory for the table. The user of the system must therefore be careful when using *switch*-statements with *case*-entries that stretch over a large range of values.

A Java compiler that generates native code can use perfect hashing here. During compilation time, a conflict-free hash function is computed which maps the case-values to target addresses. The jump is then executed in $O(1)$. As shown by Mehlhorn, such a hash table of size $3n$, where n is the number of case-labels, can be computed in cubic time [Mehlh84]. Future versions of Jamaica that generate native code will use perfect hashing here. The currently generated *C* code uses *C*'s switch statement and relies on the *C* compiler's implementation.

3.1.6 Memory Allocation

The most difficult problem to be solved by a deterministic implementation of Java is to provide deterministic behaviour of dynamic memory allocation. The implementation has to guarantee a hard upper bound for the execution time of an allocation. But this is not sufficient, it also has to guarantee that the garbage collector recycles memory sufficiently quickly for the application not to run out of memory. Finally, the implementation must guarantee that fragmentation or conservative scanning techniques do not cause the loss of memory in a way that allocations cannot be satisfied. The implementation consequently uses an exact garbage collector that has accurate information on all reference values in objects, local variables, stacks and processor registers.

To avoid fragmentation, compacting or moving garbage collection techniques are typically employed. These techniques nevertheless cause several difficulties for an efficient implementation. For example, references to moved objects must be updated to the new location of a moved object. This updating must include references in local variables or processor registers.

The Jamaica implementation uses a new approach to avoid fragmentation altogether. The heap is regarded as an array of blocks of a fixed size (typically 32 bytes per block). On allocation of an object, at least one such block is used. If this is not sufficient, a linear list of possibly non-contiguous blocks is used to represent the object. Arrays are represented as a tree of blocks with the array elements just in the leaf nodes. The use of this object and array model allows the use of non-contiguous regions of memory for allocation. There is no need to defragment memory and move objects. This technique to avoid fragmentation has been described in detail in an earlier publication [Siebert00].

In classic Java implementations, the garbage collection activities occur unpredictably whenever the system memory runs low or certain thresholds are reached. Additionally, these implementations typically cannot guarantee that the garbage collector performs sufficient recycling work to catch up with the allocation of the application. This approach is not applicable for a deterministic implementation.

Jamaica couples garbage collection activity with allocation. Whenever a block of memory is allocated, a certain number of blocks are scanned by the garbage collector. As has been shown earlier [Siebert98], this approach can guarantee sufficient garbage collection progress for the system not to run out of memory and it can guarantee upper bounds for the amount of garbage collection work that is required

k:	0.0	0.3	0.4	0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95	0.975	1.0
P_{max}:	1.0	6.173	8.085	10.67	14.47	20.71	25.67	33.10	45.45	70.11	144.0	291.8	∞

Table 1: Upper bound for required GC progress per unit of memory allocated

for the allocation of one block of memory as long as the amount of reachable memory used by the application is limited.

For an application that uses not more than a fraction k of the total heap size as reachable memory, the upper bound P_{max} of the number of units of garbage collection work that need to be performed for the allocation of one block of memory can be determined statically. Several values for k and the corresponding values of P_{max} are presented in **Table 1**. On the PowerPC architecture, one unit of garbage collection work for a block size of 32 bytes corresponds to 266 machine instructions in the worst case. This permits the determination of a worst-case execution time for the allocation time of Java objects and arrays.

3.1.7 Memory Accesses

The use of fixed size blocks to represent Java objects and arrays has an important impact on the code that is required to access object fields and array elements.

Field Accesses

Since a linear list of blocks is used to represent objects, the number of memory references required is linear in the offset of the field within the object. Fields that reside in the first block can be accessed using a single memory access, while fields in the second block require two memory accesses, etc.

Fortunately, most Java objects are very small and even for larger objects the fields that are accessed most frequently are typically the first fields with the lowest offsets [Siebert00]. For any field, the position is known statically and hence the number of memory accesses required can be determined statically.

Array Accesses

With array accesses, the situation is a bit more complex since arrays are represented as trees. The trees use the highest branching factor allowed by the block size. For a block size of 32 bytes on a 32-bit system, this means that the branching factor is 8. Even very large arrays can be represented in a fairly shallow tree. For a given system with a limited heap size h, the maximum depth d_{max} for these trees can be determined statically using the following term (for a block size of 32 bytes).

$$d_{max} = \lceil ln_8(h/32) \rceil$$

For a heap of 32MB the resulting maximal depth is 7. For an array access, one additional memory access is required to read the array's depth and one reference is needed to read the actual element. So the total number of memory references is d_{max}+2, or 9 for a heap size of 32MB.

This enables the determination of a worst-case execution time for array accesses. Even though this time is much higher than that of a classical linear array representation, it will be shown below that the overall performance of the system is comparable to that of traditional Java implementations.

3.2 Monitors

Several publications have presented efficient implementations of Java monitors that reduce the memory overhead of inlined monitors and the runtime overhead in the most common cases. Yang et. al. [Yang99] propose inlining the monitor and reserving one word per object for the monitor. This word consists of three sub-word sized integers that represent the nest count, the owner thread and a list of waiting threads. The nodes of this list are stored in a hashtable and they are only used if threads are actually waiting for a monitor.

Bacon et. al. [BKMS98] reserve 24 bits per object for the monitor. They distinguish two different representations for monitors: inlined and inflated. Inlined monitors use the monitor value to store an identifier for the owner thread and a nest count. Inflated monitors use the 24 bits as a monitor id that functions as a reference to a monitor object on the heap. Monitors that are never subject to contention remain in inlined representation, while monitors that are subject to contention will be converted to their inflated representation and will remain in this representation until the object dies.

The disadvantage of these monitor representations is their unpredictable runtime behaviour due to the use of heap allocation or a hashtable for monitors that are subject to contention.

The Jamaica monitor implementation avoids the need to dynamically allocate monitor storage on the heap or stack altogether. Monitors are always inlined, using 16 or 32 bits per object (16 bits are sufficient in systems with few threads and a limited nest count). As long as no threads are waiting for a monitor, it is sufficient to record the owning thread's identifier and the nest count in the inlined monitor (**Figure 4**). As soon as a thread tries to acquire a monitor that is owned by a different thread, a queue of waiting threads needs to be created.

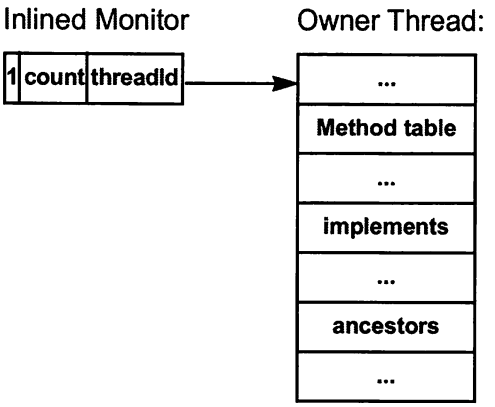


Figure 4: Inlined monitor owned by one thread.

An important observation one can make is that no thread can ever be waiting for more than a single monitor. Hence, instead of creating independent node objects for the waiting queue, one might as well reserve some additional fields in the thread object itself and create a queue of thread objects for all waiting threads. Since a thread might own several monitors simultaneously, the owning thread itself must not be part of this queue, only the waiting threads can be linked to the monitor. One bit in the inlined monitor indicates that a waiting queue exists. In this case, the nest count and owning thread's identifier are copied into reserved fields of the threads in the waiting queue. The identifier of the first thread in the queue is stored in the inlined monitor (**Figure 5**).

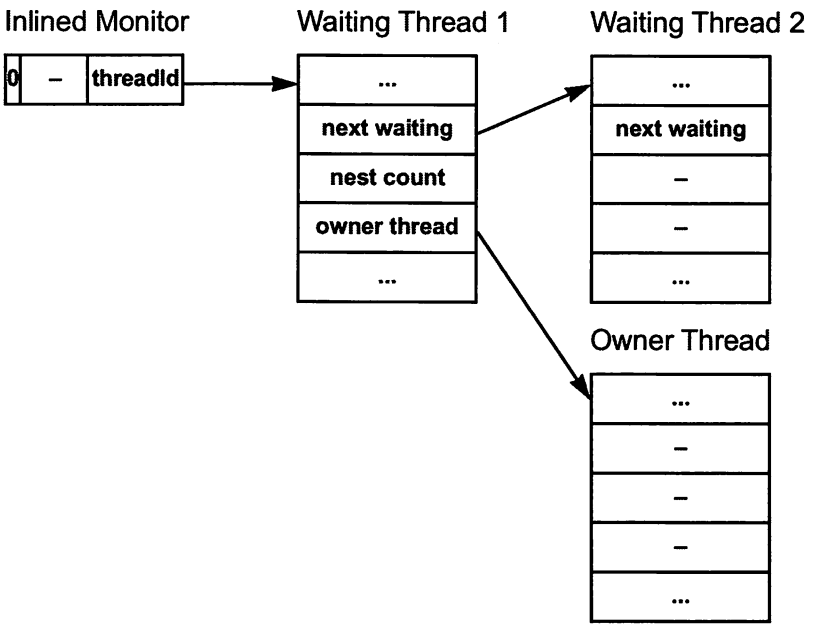


Figure 5: Monitor owned by one thread with two waiting threads.

The result is a monitor implementation that is very efficient for the most frequent cases of monitor operations. Entering a monitor that is not owned by any thread or that has already been acquired by the current thread and exiting a monitor that has no other threads waiting. The operations that involve waiting threads do not require dynamic creation of a monitor object, all that needs to be done is queuing and unqueuing the waiting threads and performing the actual wait or resume.

3.3 Exceptions

Java's exception mechanism enables control flow from an active method back to a method with an exception handler that lies an arbitrary number of stack frames above the current position in the call chain. Throwing an exception that is handled by another method therefore requires removal of all active stack frames that lie in between the method causing the exception and the method handling it. This includes removing of all local references in these stack frames from the garbage collector's root set, which is an

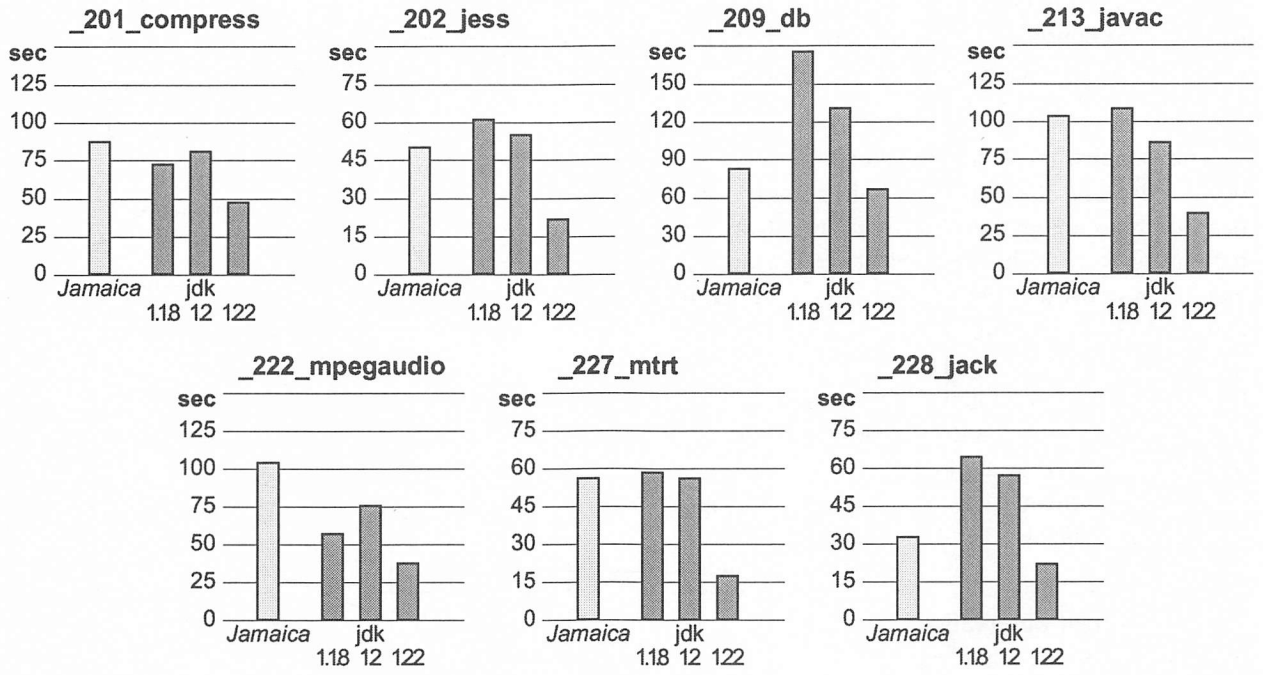


Figure 6: Runtime performance of the SPECjvm98 benchmarks using Jamaica and JDK 1.1.8, 1.2 and 1.2.2

operation linear in the number of active stack frames and in the number of root references.

The execution time for an exception is therefore not constant and it is at least difficult to find a constant-time implementation for exception handling. Using a mechanism like *C*'s *setjmp* and *longjmp* enables constant-time change of the control flow that is required for exceptions, but it does not provide means to remove the information required for accurate garbage collection.

In the Jamaica implementation, the execution time of throwing an exception is therefore linear in the size of the stack frames between the exception throwing point and the corresponding exception handler. Since exceptions are not supposed to be used for normal control flow, but for exceptional control flow only, this behaviour should be acceptable even when deterministic execution in the non-exceptional case is required.

4. PERFORMANCE COMPARISON

To evaluate the overall performance of the deterministic Java implementation Jamaica and to compare it to traditional Java implementations, the SPECjvm98 benchmark suite [SPEC98] has been run using Jamaica and several versions of SUN's JDK [SUN99, SUN00] with JIT compiler enabled (*JDK 1.1.8*; *Classic VM build JDK-1.2-V, green threads, sunwjit*; and *Solaris VM build Solaris_JDK_1.2.2_05, native threads, sunwjit*). Only one test from the benchmark suite, *_200_check*, was excluded from the analysis since it is not intended for performance measurements but to check the correctness of the implementation (and Jamaica passes this test).

For execution, the test programs were compiled and smart linked using the Jamaica builder. The programs were then executed on a single processor (333 MHz UltraSPARC-IIi) SUN Ultra 5/10 machine equipped with 256MB of RAM running SunOS 5.7.

The results of the performance measurements are shown in **Figure 6**. For the measurements, the heap size was set to 32MB for all tests but *_201_compress*. This test required more memory to execute and was run with a heap size of 64MB.

Compared to Sun's implementation, the performance of the deterministic implementation is similar to that of JDK 1.1.8 or 1.2, while the performance of JDK 1.2.2 was improved significantly. Since Jamaica is still a very young implementation and not much effort has been spent on improving and tuning the

compiler's optimization techniques, one can expect that better optimization in the compiler will permit to improve the performance of Jamaica further. Another source for future performance enhancements will be direct generation of machine code instead of using *C* as an intermediate language as is done currently. Using *C* does not permit optimal code selection for primitives like write-barrier code or garbage collector root reference information. Direct generation of machine code permits selection of better code for these primitives, e.g., by assigning certain registers for specific purposes.

5. CONCLUSIONS

In this paper the Jamaica implementation of Java has been presented. The deterministic implementation of Java's primitive operations in this implementation has been explained. The implementation permits static analysis of the execution time of its primitive operations that is not possible with current Java implementations.

The performance of the implementation has then been measured and compared to different versions of Sun's JDK implementation using the SPECjvm98 benchmark suite. The results show that performance that is comparable to Sun's implementations can be reached. These results encourage us to further improve our implementation and to further reduce the performance gap compared to non-deterministic Java implementations.

6. FUTURE WORK

The deterministic implementation of a programming language's primitive operations provides only the basis for further analysis of the code. Tools for automatic determination of worst-case execution times can be build on top of this. For an accurate analysis that is not too conservative, mechanisms to model the system's cache memories and the effects of modern superscalar processors on the execution time need to be developed.

7. AVAILABILITY

The described Java implementation is available for academic and non-academic purposes. Please contact the authors or visit the web-site at *http://www.aicas.com* to obtain further information.

REFERENCES

[AG98] Ken Arnold and James Gosling: *The Java Programming Language*, 2nd edition, Addison Wesley, 1998

[BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, Mauricio Serrano: *Thin Locks: Featherweight Synchronization for Java*, PLDI, 1998

[DS84] L. Peter Deutsch and Allan M. Schiffman: *Efficient Implementation of the Smalltalk-80 System*, Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, pp. 297-302, Salt Lake City, UT, January, 1984

[JCons99] J-Consortium: *Real Time Core Extension for the Java Platform*, Draft International J-Consortium Specification, V1.0.14, September 2, 2000

[LL73] C. L. Liu and James W. Layland: *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, Journal of the ACM, Vol. 20, No. 1, January 1973, pp. 46-61

[LY99] Tim Lindholm and Frank Yellin: *The Java Virtual Machine Specification*, Second Edition, Sun Microsystems, 1999

[Mehlh84] Kurt Mehlhorn: *Data Stuctures and Algorithms 1: Sorting and Searching*, Springer Verlag, Berlin, 1984

[Meyer92] Bertrand Meyer: *Eiffel: The Language*, Prentice Hall International (UK), Hertforshire, 1992

[Nilsen96] Kelvin Nilsen: *Java for Real-Time*, Real-Time Systems, 11, pp. 197-205, Boston, 1996

[NR95] K.D. Nilsen and B. Rygg: *Worst-Case Execution Time Analysis on Modern processors*, ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, San Diego, 1995

[RTJEG99] The Real-Time Java Experts Group: *Real Time Specification for Java*, Addison-Wesley, 2000, http://www.rtj.org

[Siebert98] Fridtjof Siebert: *Guaranteeing non-disruptiveness and real-time Deadlines in*

an Incremental Garbage Collector (corrected version), International Symposium on Memory Management (ISMM'98), Vancouver, 1998, corrected version available at *http://www.fridi.de*

[Siebert00] Fridtjof Siebert: *Eliminating External Fragmentation in a Non-Moving Garbage Collector for Java*, Compilers, Architectures and Synthesis of Embedded Systems (CASES'00), San Jose, November 2000

[Strou87] Bjarne Stroustrup: *Multiple Inheritance for C++*, Proceedings of the European Unix Users Group Conference, pp. 189-207, Helsinki, May, 1987

[SPEC98] *SPECjvm98 benchmarks suite, V1.03*, Standard Performance Evaluation Corporation, July 30, 1998

[SUN99] *Java Development Kit 1.1.8*, SUN Microsystems Inc., 1999

[SUN00] *Java Development Kit 1.2.2*, SUN Microsystems Inc., 2000

[Weiss98] Michael Weiss, François de Ferrière, Bertrand Delsart, Christian Fabre, Frederick Hirsch, E. Andrew Johnson, Vanial Joloboff, Fred Roy, Fridtjof Siebert, and Xavier Spengler: *TurboJ, a Bytecode-to-Native Compiler*, Languages, Compilers, and Tools for Embedded Systems (LCTES'98), Montreal, in Lecture Notes in Computer Science 1474, Springer, June 1998

[Yang99] Byung-Sun Yang, Junpyo Lee, Jinpyo Park, Soo-Mook Moon, Kemal Ebcioglu, Erik Altmann: *Leightweight Monitor for Java VM*, ACM Computer Architecture News, Vol 27/1, March 1999.

[ZCC97] Olivier Zendra, Dominique Colnet and Suzanne Collin: *Efficient Dynamic Dispatch without Virtual Function Tables*, OOPSLA, 1997

Mostly Accurate Stack Scanning

Katherine Barabash*
kathy@il.ibm.com

Niv Buchbinder*
nivb@il.ibm.com

Tamar Domani*
tamar@il.ibm.com

Elliot K. Kolodner*
kolodner@il.ibm.com

Yoav Ossia*
yossia@il.ibm.com

Shlomit S. Pinter*
shlomit@il.ibm.com

Janice Shepherd†
janshep@us.ibm.com

Ron Sivan*
rsivan@il.ibm.com

Victor Umansky‡
victor@sangate.com

* IBM Haifa Research Laboratory † IBM T. J. Watson Reasearch Laboratory ‡ Sangate Israel

Abstract: The Java Virtual Machine (Jvm) needs, for the purpose of garbage collection (GC), to determine the data type stored in every memory location. Jvms that can do this reliably are said to be *type-accurate* (TA). Full type-accuracy usually exacts a price in performance due to the need to scan stacks and registers accurately. The mostly accurate approach presented here can reduce the TA overhead significantly by sacrificing accuracy for the small minority of memory locations that add the most to the cost. Performance results show that mostly accurate stack scanning performs as well as conservative stack scanning and that relatively few objects are identified conservatively.

In addition our implementation is designed to support and generate type maps for any verifiable bytecode stream (including combinations that are unlikely to be produced by a compiler) without requiring rewriting of the bytecode. We introduce a new compression technique for type maps that uses a program-friendly format for the maps; yet, achieves good compression and provides fast opening of compressed maps. We show how to apply systematic testing techniques and test coverage tools to an accurate stack scanner.

Keywords: Jvm, type accuracy, garbage collection.

1 Introduction

For Java Virtual Machines (Jvms) the unequivocal identification of references at run time may be difficult. This is mainly a problem for the task of Garbage Collection (GC), mandated by the Jvm Specification [29], which needs to make such determinations.

In a commonly used solution, known as conservative GC [10], any value that could possibly signify a reference is treated as one; this includes both values in objects and values on the stack. While avoiding the danger of ever missing a reference, conservative GC runs the risk of considering values that are actually integers or irrelevant, thus retaining garbage. Moreover, this approach is incompatible with schemes that move objects, since when a value used as a reference is not guaranteed to be one, updating it (as would be necessary if the object it points to is relocated) could have disastrous effect if in fact the value was not a reference.

Another approach, often described as conservative with respect to the roots [8], scans the stacks and registers conservatively, but uses type information associated with the objects to scan them accurately. This approach is used widely in Java Virtual Machines, where the run time is implemented in C, e.g., the Sun Classic JVM [12] and IBM's Developer Kit [25]. In these Jvms scanning the stacks accurately is difficult, as C compilers do not typically produce type information for the stack frames they generate. However, type information is read-

ily available for object slots. The conservative with respect to the roots approach, while retaining less garbage and allowing object relocation, complicates the use of copying GC [11], compaction, and algorithms such as generational scavenging [27, 40, 32] and Train [24, 36], which rely on copying. Nevertheless, there is a range of GC techniques available that do not require that all objects be relocatable, e.g., [8, 9, 33, 15, 13, 16, 35].

A third approach is type-accuracy (TA) [14], where the location of all references, both in the roots (stacks and registers) and objects, can be determined with certainty. TA for stacks and registers is generally obtained by producing type maps for methods, and using these maps to interpret stack frames as they are scanned by GC. This approach is also widely used in Jvm implementations [23, 5, 17, 26]. It is often considered the most technologically advanced of the three approaches since it allows all objects to be relocated and retains the least amount of garbage.

As appealing as TA sounds, it comes with a performance and a complexity price, especially for Java in server environments where applications are highly multithreaded and may make frequent transitions into native code, e.g., to enable access to legacy code. In a multithreaded environment all threads must be stopped for GC at *safe points*, execution points where type information for volatile structures such as stack frames and registers is available. GC safe points are typically placed at invocation sites and on backward branches in loops. Thread stopping is usually done through polling or code patching. Agesen [2] found that polling has a high performance overhead and that code patching is more efficient for single-threaded programs. However, code patching is complex and its performance is not likely to scale in a highly multithreaded environment, where every thread may potentially need to be patched. Stichnot et al. [39] report that the amount of space required to save type information for every compiler generated instruction for Java code can be kept manageable using appropriate compression techniques. This would avoid stopping overhead by allowing a thread to stop at any point in Java code. However, it remains to be shown that this approach performs well, as the compression/decompression algorithms are likely to be time consuming, and that it extends to handle native code where no maps are available.

Transitions from Java methods into native code require the saving of non-volatile registers. This is because compilers for native code, e.g., for C, do not generally keep track of type information or where registers are saved in stack frames. Frequent saving of non-volatile registers may also hurt performance. Dealing with registers may also present other problems. In particular, not all platforms allow the registers of a stopped thread to be updated by a non-privileged thread. This may be required in case registers contain references to objects that have been relocated.

1.1 Our Contribution

We propose a fourth approach, that is mostly accurate with respect to the roots. It is a pragmatic approach that has the potential to deliver a high degree of type accuracy, thereby reducing retained garbage and the number of non-relocatable objects, while incurring little or no performance overhead as compared to conservative stack scanning. It is also significantly less complex than full accuracy, as it does not require complicated schemes for stopping threads or for identifying all frames on the stack.

In principle, mostly accurate stack scanning applies accurate scanning only to frames where it is easy and fast to do so, and scans the remaining frames conservatively. Thus, threads are no longer required to be stopped at GC safe points but may be stopped anywhere, at the price of a conservative scan of the most recent frame on the thread's stack. Most other frames are expected to be at invocation points and therefore remain candidates for accurate scanning. Of those, frames whose type information is complicated to obtain (e.g., native code frames) are also scanned conservatively. Our implementation also does not pay the cost for saving non-volatile registers before transitions into native code. This does not incur an additional loss of accuracy since native code frames are scanned conservatively in any case.

We implemented a mostly accurate scanner, replacing the conservative stack scanner in a prototype version of IBM's Developer Kit for Windows NT that is compatible with Java 2. The GC in this Jvm is a stop-the-world parallel mark and sweep collector, which compacts only when necessary. We compared the performance of the mostly accurate scanner with the conservative scanner. Our results confirm that mostly accurate scanning performs as fast as conservative scanning and reduces the number of objects that are identified conservatively.

We also used several interesting techniques applicable to other implementations of type-accuracy. Our implementation of accurate stack scanning is designed to support and generate type maps for any verifiable bytecode stream (including combinations that are unlikely to be produced by a compiler) without requiring rewriting of the bytecode. It is based on the scheme first used by Jalapeño [6, 5]. In addition, we introduce a new compression technique for type maps that uses a program-friendly format for the maps; yet, achieves good compression and provides fast opening of compressed maps. Finally, we show how to apply systematic testing techniques and test coverage tools to an accurate stack scanner.

1.2 Outline

Section 2 presents our overall approach to scanning stack frames accurately and describes which frames we scan accurately and which frames we scan conservatively. Section 3 discusses map generation for interpreted Java methods. Section 4 describes how we store and compress the type maps, and provides performance results for our compression scheme. Section 5 shows how systematic testing techniques can be applied to accurate stack scanning. Section 6 presents performance results for our prototype implementation of mostly-accurate stack scanning. Finally we conclude in Section 7.

2 Mechanics of Type Accuracy

The task of running a Java program can be split in the Jvm into several subtasks that together do the job correctly and efficiently. Type accuracy is achieved by a combination of means and techniques which collectively cover the various subtasks and the many cases within them that need to be handled. First we briefly review the mechanics of TA in general. Then we describe which parts of the stack we scan accurately and which conservatively.

2.1 Type Accuracy

References to objects can occur within other objects or in volatile execution structures: stack frames and machine registers. References within an object are documented by the class of the object. GC uses this information recorded in the class to accurately scan object instances. Comparable descriptions of volatile structures must be generated specifically to support TA. These descriptions often take the form of *type maps* which indicate which stack slots and machine registers contain a reference at a given execution point of a given method.

The Jvm subtasks may be broadly categorized as follows:

- Interpretation of Java bytecode.
- Execution of Just-In-Time compiled Java code (JITted code).
- Execution of non-Java (native) code.
- Jvm services, such as object allocation, class loading, JIT compilation and GC.

Type maps for frames of interpreted bytecode are generated by analyzing the bytecode itself; we describe map generation in detail in Section 3. Type maps for compiled code are generated by the JIT compiler as part of its compilation process. Non-Java methods (native code) are expected to abide the Java Native Interface (JNI, [30, 31]) that closely controls the usage of references. References are generally kept outside the reach of native code, and JNI provides mechanisms for tracking them in the few exceptions where they are not. Finally there are Jvm services, which are routines written in the implementation language of the Jvm, in our case, C.

Scanning of volatile structures is done using the following elements:

1. stack scanner
2. frame traverser
3. type map repository
4. type map generator

GC calls upon the stack scanner to scan the stacks. The stack scanner resorts to the frame traverser to walk the stack one frame at a time and to determine its particulars: which of the subtasks above owns the frame, what method is it executing and where is the execution of the method stopped. Based on

the method and the point within its execution, the stack scanner attempts to obtain a type map for the frame from the repository. Type maps are normally available for compiled methods since the JIT compiler deposits maps for every method it compiles. If the method is interpreted and its maps are not yet available in the repository, the stack scanner invokes the map generator to create maps for the method and then saves them in the repository for possible future use. Note that in the case of interpreted code, this arrangement results in maps being calculated only for methods that are active at the time of GC. Once a map is available, the stack scanner can identify the slots containing references and report them to GC.

2.2 Mostly Accurate Stack Scanning

The frame traverser is able to recognize stack frames for Java methods, whether interpreted or JITted. Accordingly, our mostly accurate stack scanner scans all frames belonging to interpreted and JITted methods accurately, except for the most recent frame on the stack of each stopped thread. All other frames, which are just the regions on the stack between groups of Java frames, are scanned conservatively. Thus, the stack scanner does not differentiate between frames belonging to native methods and Jvm services.

3 Generating Type Maps

The type map generator for the interpreter analyzes the bytecode of a method and produces maps for any execution point at which GC could occur. Below we provide an overview of the type map generator. We adopted the approach used by Jalapeño [5], which does not require rewriting of bytecode. Then we briefly discuss maps for JITted methods.

Since the use of slots in a stack frame could change during the course of execution of a method, maps depend on the point of execution. Conceivably, the layout of the frame need not be unique even at a specific execution point, but could depend on the history of execution. However, due to the *Gosling property* [19, 4], which holds for Java bytecode, except for `jsr` subroutines (discussed below), and is verified at class load time, the stack frame structure is normally independent of the execution path. This allows the use of a basic algorithm, similar to the one used for verification[29, Section 4.9], consisting of three steps:

1. The bytecode for a method is split into basic blocks.
2. An iterative algorithm computes the type map for the start of each basic block.
3. The maps for the GC points (e.g., invocation points, allocation points) within each basic block are calculated.

3.1 jsr Subroutines

The `jsr` and the accompanying `ret` bytecodes are a source of difficulty for generating type maps since their semantics permits a violation of the Gosling property. This bytecode pair is used to declare an inner subroutine within a method, which does not have its own frame. Java compilers use `jsr` subroutines to control the flow from a sequence with many exits into a sequence that must be executed regardless of the exit taken, e.g., a `finally` clause, or the release of the lock at the end of a `synchronized` method. Semantically, `jsr` is an intra-method branch instruction just like the `goto` bytecode, except that it also pushes a return address (the offset of the bytecode following the `jsr`) onto the operand stack. If that address is subsequently stored in some local variable, the `ret` bytecode can be used to jump back to the point from which the subroutine was called.

Various restrictions apply to `jsr` subroutines: they may be entered only via a `jsr` and may not call themselves recursively. However, there is no restriction on the contents of local variables that the subroutine does *not* reference. As a result, the same local could contain a reference at some invocations of the `jsr` subroutine and a non-reference at others (see Figure 1). This will not disturb the execution of the subroutine, but if it is stopped for GC and the types of its frame slots are needed, the state of that local could be indeterminate.

3.2 Solution For jsr

A solution, adopted here from the work done on the Jalapeño project [5], postpones the generation

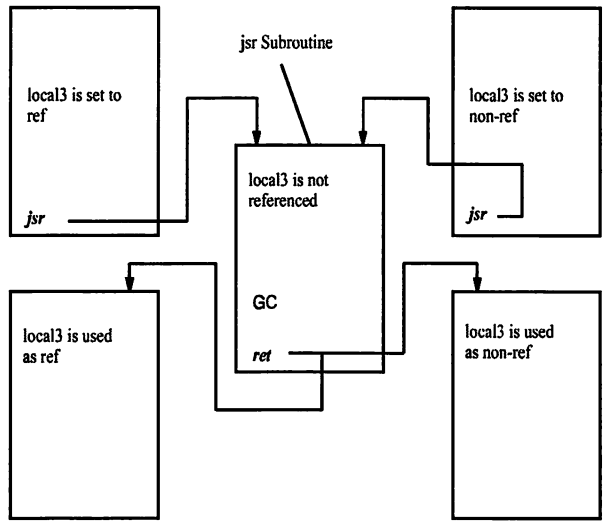


Figure 1: `jsr` subroutine where the Gosling property could be violated: the contents of `local3` at the time of GC may depend on the history of execution.

merge a change map	with a final map		with another change map		
	non-ref	ref	to-non-ref	to-ref	neither
to-non-ref	non-ref	non-ref	to-non-ref	to-ref	to-non-ref
to-ref	ref	ref	to-non-ref	to-ref	to-ref
neither	non-ref	ref	to-non-ref	to-ref	neither

(a) **(b)**

Table 1: Rules for merging a change map with (a) a final map and (b) with another change map

of the map until run time, when the actual path the execution has taken is known. The generator prepares two kinds of maps in advance: *final maps* and *change maps*. A final map is a bit array where each bit is associated with a stack frame slot. Slots expected to contain a reference (`ref`) have their bits set. A change map consists of two bit arrays that together represent the difference between two states of the stack frame, and hence between two final maps. One array, `to-ref`, indicates the locations that have changed to contain a reference, and the other, `to-non-ref`, indicates changes to a non-reference. A change map C can be *merged* with a final map F to obtain a new final map F', representing the state of a stack frame after applying the change described in C to the state described in F (see Table 1 **(a)**). Two change maps can also be merged, producing a new change map which describes the result of applying them consecutively (see Table 1 **(b)**. Note that this merge operation is not commutative.)

Within `jsr` subroutines the map generator calculates change maps, which reflect the changes from the beginning of the subroutine to the point for which the map was made. In addition, the map contains the location at which the return address provided by the `jsr` is kept. At run time, the Jvm retrieves the return address and uses its actual value to determine which `jsr` actually invoked the present subroutine, and obtains another map from there. If the latter map is a final map, merging the two maps yields a final map which correctly describes the present frame. If the map at the site of the `jsr` is a change map as well, (probably due to nesting of `jsr` subroutines), merging the maps produces yet another change map. The procedure is repeated recursively until a final map is obtained.

Finding the return address is not always simple. Normally, a `jsr` subroutine begins by storing the return address in some local and using that local in the `ret` at the end. However, verifiable code in general need not be this simple, and the return address could at times be rather elusive:

- A subroutine may store the return address in a local, but not at the very beginning of its execution. Consequently, the return address may still be on the operand stack when a map is required.
- The subroutine may dup the return address and use the duplicate, rather than the original value, for returning.
- The subroutine may discard the return address altogether, but then it may no longer return, and in effect it is no longer a subroutine.
- In case of nested subroutines, there are two return addresses in effect at the same time. The inner subroutine may swap the two values if they are adjacent on the operand stack. This could confuse the recursion process described above, as the map for the outer subroutine could specify a return address location, which is not the right one after the swap. This is an esoteric case that was discovered during the course of map testing, which is described in Section 5.

Much of the complexity in implementing the map generator centered on the tracking of the return addresses for `jsr` subroutines.

3.3 Maps for Compiled Methods

Maps are generated for compiled methods as well. Unlike interpreted methods that are produced on demand, maps for compiled methods are created by the JIT compiler as a byproduct of the compilation. The maps are then kept in the repository until they are needed. Producing them on demand would essentially require recompiling the method and would be too costly.

Compiled methods also have to deal with various aspects of the architecture that are shielded by the interpreter. Non-volatile registers (NVRs) are one such aspect. To abide by the calling convention, a method must save an NVR in its frame before its first use of the NVR and restore the NVR before returning. However, the method saving the NVR cannot know the type of its value. To overcome this problem, each map contains information both about what values its method places in NVRs on the one hand, and which NVRs it saves and where on the other. The stack scanner can intersect content information from one map with storage information from another as it traverses the stack, and so determine which slots used for saving NVRs actually contain references.

Another issue is that of *untidy references*, i.e., references which do not point to the top of objects as canonical, or *tidy*, references do. Untidy references are generated by the compiler when optimizing access to arrays, for example. Each type map contains a list of all such references if any exist in the frame, together with how they are dependent on the corresponding tidy references (see [14]). The stack scanner is again responsible for updating any untidy references whenever any of the tidy ones they depend on get modified in the process of GC. Our current implementation does not handle untidy references yet and instead scans them conservatively.

When compiled methods are concerned, the normal relationship of one frame per each active method may no longer hold. For example, there are cases where additional frames are injected between those of otherwise adjacent methods. This is done when incompatible calling conventions require reformatting of the arguments, or when a method switches from being interpreted to being compiled. These extra frames may also contain references, but for the most part are currently scanned conservatively.

3.4 Related Work

The challenge of generating type maps in the presence of `jsr` was first described by Ageson *et al.* [3, 4]. They solve the problem by splitting the problematic variables in two, one for the reference type and the other for the non-reference type. This requires rewriting of bytecode. Since there are size restrictions on a method's stack and the bytecode, this solution will not always work.

Another solution [1] maintains tags for ambiguous variables that indicate their current type at run

time. This requires code to initialize and maintain the tags, and incurs an overhead upon method invocation and with each update of the affected variables.

A third approach [39] is similar to the one used in Jalapeño [5], which was described above. It also makes compile-time information available to the Jvm, which can then apply run-time knowledge to determine the true type of ambiguous variables.

4 Repository for Maps

Generation of type maps for stack frames raises the need for map management. Although maps could be generated on demand each time one is required, such an approach would be wasteful and would hurt performance. A map, once created, is kept in a repository for possible future use. This section describes the repository and the compression technique it uses.

4.1 Requirements

Maps for interpreted methods are calculated on demand. Maps for JITted methods are produced as a byproduct of the compilation process. Each method may have multiple maps associated with it that are all produced at one time.

Type maps, in addition to indicating which slots of the frame contain references, provide information necessary for proper interpretation of the frame at run time. Some fields of the map fit in fixed sized fields, such as the number of local variables or operand stack entries, while others, such as the list of untidy references in JITted maps, are of variable length. The maps consist of a fixed size header, followed by a variable length data area, whose components and size are derivable from information stored in the header.

When interpreting a frame, only one of the maps of the corresponding method is ever needed. Over time, however, several or even all of the method's maps may be called for, particularly for methods that are called often. On the other hand, methods such as "main", that are called shortly after the run starts and remain active for a long time, may have their frames examined several times, requiring the same map over and over again.

In summary, the characteristics of map usage make the following demands on the repository:

- Maps should be saved for future use, avoiding the overhead of recalculating them. There are maps for all JITted methods and for all interpreted methods for which a map was ever requested. This could add up to a large number of maps, so compression, efficient in both time and space, is required.
- Maps are called for in random order, but only one map per method at a time. This requirement of fast retrieval of randomly selected maps favors compression schemes where each map is compressed separately from the others.
- Maps should not require a lot of bit operations for extracting information. This simplifies the code of the stack scanner. This means that maps will be wider, but we need a compression scheme in any case.

We expect the maps to have the following characteristics:

1. Methods may have any number of maps, including none.
2. Most of the methods will have only a small number of maps.
3. Maps of the same method should be similar to each other, yet there is no part of all the maps of a method that is guaranteed to be the same.
4. For methods with many maps, the similarity should increase with proximity (within the bytecode stream), i.e., maps of neighboring bytecodes should be more similar than maps associated with bytecodes that are further apart.
5. Most of the maps are not big, less than 50 bytes on average.
6. All of the above holds true for both JITted and interpreted methods.

Examination of actual maps corroborated most of the assumptions above. Assumptions 2 and 4 did

not quite hold for interpreted methods in short-lived applications. The reason could be that since maps for interpreted methods are created on demand, longer methods which invoke many additional methods are more likely to be encountered on the stack and therefore over-represented among the maps. Indeed, the failure of these assumptions was less pronounced in longer-running applications, where eventually maps were created for more of the methods.

4.2 Compression Techniques

In order to meet the requirements above, and with consideration of the characteristics of maps, we chose to represent every map as the difference from a specific (fixed) map of the same method. The latter was called the *pivot map*. A variation on the run-length encoding scheme [18], which we call *dirty runs*, is used to record the difference between a map and the pivot map.

A compressed map consists of a sequence of runs, $\{R_1 \ldots R_n\}$. Each run R_i has the structure $\{S_i, D_i, B_1 \ldots B_{D_i}\}$. S_i is the number of similar bytes, D_i is the number of different bytes that follow, and $\{B_1 \ldots B_{D_i}\}$ are the actual D_i different bytes. Maps are assumed to start with similar bytes (if this is not the case, S_0 is zero), and any tail part of the compressed map that is not accounted for in the runs is assumed to be similar to the tail of the pivot map.

Considering the expected size of maps, the S and D fields are almost always less than 128 and thus representable in 7 bits. The sign bit is therefore free to indicate a further optimization: if $S_i \leq 15$ and $D_i \leq 7$ then both values are packed into a single byte whose sign bit is set to indicate that such compression was done.

Maps turn out to contain many zero bytes, so the pivot map itself is saved internally as compressed against a map of similar length containing only zeros. This proved to be a good practice, even for methods having only one map.

The maps relating to one method are combined into a stream of bytes, consisting of:

- A small header indicating the number of maps and the index of the pivot map.
- A "table of contents" (TOC) listing the compressed size of each map. TOC entries are all the same size, but may differ from method to method, and are large enough to accommodate the largest value for the method.
- The dirty runs of all the maps of the method

4.3 Selecting the Pivot Map

The representation of a map is shorter the more similar the map is to the pivot. Thus, choosing a good pivot map is central to the quality of the compression on the whole.

Given a set of N maps, an exhaustive search for the best pivot map among them requires N^2 different compression attempts. We used a heuristic to select a good pivot map in N steps.

Let M_p be the pivot map and M_c a map to be compressed. Define $Cost(M_p, M_c)$ as the length of the compressed representation of M_c with respect to M_p. (With dirty runs scheme, the cost is the number of S and D bytes plus the number of bytes that are actually different between the maps). Define further $TotalCost(M_p)$ as the total length of the compressed representation of all the maps in the method:

$$TotalCost(M_p) = \sum_{i=1}^{N} Cost(M_p, M_i) \qquad (1)$$

where N is the number of maps in the set. The purpose is to find a p such that $TotalCost(M_p)$ is minimal. The algorithm is based on the heuristic assumption that if:

$$Cost(M_p, M_1) - Cost(M_p, M_2) = \delta \qquad (2)$$

then

$$Cost(M_2, M_1) \approx \delta \qquad (3)$$

In other words, a subset of maps that are different from a pivot map by similar degrees will probably be compressed at a lower cost if one of them is chosen as the pivot map instead. It is therefore possible to predict the results of compression with one pivot map based on the actual result of compressing it with another.

The algorithm for choosing a good pivot map is as follows:

1. A map in the set is arbitrarily selected as the initial pivot map M_{init}. (Cost: $O(1)$.)

2. All maps of the set are compressed using M_{init} as pivot. (Cost: $O(N)$.)

3. A histogram H of the costs of all the compressions of maps of the set is created. $H(\lambda)$ is the number of maps whose length after compression is λ, i.e.,

$$H(\lambda) = \| \{i \mid Cost(M_i, M_{init}) = \lambda\} \| \quad (4)$$

For each λ for which $H(\lambda) > 0$, a pointer to a map whose cost is λ is kept. The histogram itself is of size K, where K is the longest compressed representation of any of the maps in the set. With dirty runs, K is usually much less than the maximal uncompressed string length. (Cost: $O(N)$ in time and $O(K)$ in space.)

4. A λ_{opt} is extracted from the histogram H. In our implementation, the expected λ is used:

$$\lambda_{opt} = \frac{\sum_{\lambda=1}^{K}(\lambda \times H(\lambda))}{N} \quad (5)$$

(Cost: $O(K)$.)

5. A map M_{opt} associated with λ_{opt} is selected as the pivot map. (Cost: $O(1)$ since no search is involved.)

The compression scheme allows for more than one pivot map. This capability may be used to improve the overall compression ratio. Each map is then compressed using the pivot it is most similar to, thus reducing the space for its representation. This approach is useful when the map population is big and aggregated in a small number of clusters. There is a cost involved in encoding the pivot used for compressing each map into its representation. We chose to allow a maximum of two pivot maps, extended the header to include the actual serial numbers of those pivot maps, and used a bit in every field of the TOC to record the logical index $\in$ [0..1] of the pivot map used in that compression.

To determine whether using a second pivot map is advantageous for a given map set, as well as selecting such a second pivot appropriately without repeating the expensive step 2 above, the algorithm is modified as follows:

- Step 4': The λ_{opt} is selected differently in this case to be at the center of the largest cluster in H. (Cost: $O(K)$.)
- Step 5': Same as 5 above.
- Step 6': A second histogram H_2, estimating the effect of using M_{opt} as pivot, is calculated:

$$H_2(\lambda) = H(\lambda_{opt} - \lambda) + H(\lambda_{opt} + \lambda) \quad (6)$$

One of the maps that was associated with either $\lambda_{opt} - \lambda$ or $\lambda_{opt} + \lambda$ is now made to represent λ. H_2 is smaller in size than H. (cost: $O(K)$.)

Steps 4 and 5 above are repeated using H_2 to obtain a secondary (and subsequent) pivot maps.

4.4 Compression Results

The algorithm above was tested with SPECjvm98 [38]. The tests were run in two modes:

1. Interpreter only mode, where the JIT compiler is not active.
2. Mixed mode, where methods are interpreted initially, and are JIT-compiled only if they are invoked sufficiently often.

Compression statistics collected from tests in the two modes above are summarized in tables 2 and 4, respectively. The numbers in Table 4 represent the sum of the resources taken by the Jvm while running the test (e.g., "total maps size" is the total size of all maps generated, either for interpreted or for JITted methods; maps of same method may appear in both.).

There are a few observations that can be made about these results:

- The compression ratio is computed as the ratio between the space taken by the maps (after compression) and the size of the method's code (bytecode if interpreted, object code if JITted). We see that in all practical cases the space required for maps is less than that taken by the method's code.
- Comparing corresponding entries in Tables 2 and 4, it can be seen that the compression ratio is improved when JITting is done. This is not due to smaller map size but larger code

test name	number of methods	total size of code (C)	maps total raw size	maps total compressed size (M)	M/C	improvement optimal pivot	improvement two pivots
compress	17	2689	10552	3293	1.22	6.0%	10.9%
jess	42	5682	20212	6233	1.10	5.3%	9.1%
db	22	3204	11672	3655	1.14	6.5%	9.7%
javac	121	21599	64376	21696	1.00	8.1%	7.5%
mpegaudio	12	2114	8592	2615	1.24	3.8%	11.5%
mtrt	27	4793	19120	5972	1.25	8.8%	8.4%
jack	59	10957	36424	10281	0.94	6.1%	8.4%
total	300	51038	170948	53745	1.05	7.0%	8.5%

Table 2: Compression results for SPECjvm tests using bytecode interpretation only.

test name	number of methods	total size of code (C)	maps total raw size	maps total compressed size (M)	M/C
compress	222	16757	54352	12054	0.72
jess	841	50996	175900	51392	1.01
db	212	16988	60456	13802	0.81
javac	1309	102703	255204	77430	0.75
mpegaudio	489	65257	82296	20109	0.31
mtrt	354	25671	90528	23106	0.90
jack	490	48819	123892	31427	0.64
total	3917	327191	842628	229320	0.70

Table 3: Compression results for SPECjvm tests using bytecode interpretation only, when maps are generated for all methods (rather than only for those caught on the stack at GC time).

test name	number of methods	total size of code (C)	maps total raw size	maps total compressed size (M)	M/C	improvement optimal pivot	improvement two pivots
compress	32	6446	17000	4665	0.72	3.7%	4.9%
jess	139	41395	111288	26512	0.64	3.6%	3.0%
db	56	8513	26424	7090	0.83	4.6%	4.4%
javac	517	189342	365320	81113	0.43	2.8%	1.6%
mpegaudio	97	11274	36448	9206	0.82	3.1%	2.9%
mtrt	93	33448	72496	15269	0.46	2.9%	2.4%
jack	288	117225	243376	53097	0.45	3.1%	4.1%
total	1222	407643	872352	196952	0.48	3.1%	2.8%

Table 4: Compression results for SPECjvm tests using both bytecode interpretation and compilation (JITting).

size: bytecode is a rather terse program representation, and is typically much smaller than the object code created by the JIT compiler for the same method.

- The last two columns in Tables 2 and 4 indicate the improvement in compression ratio (in percentage points) that could be achieved if one of the alternatives described above were used instead. The column labeled "optimal pivot" applies to compression using the very best map of the set as a pivot. Note that finding this optimal pivot requires $O(N^2)$ compression tests for a set of N maps. The column labeled "two pivots" applies to the second option, which provides two good pivots without recalculating the histogram.

4.5 Related Work

Space for type maps is an issue every system that makes use of them must deal with. Agesen *et al.* [4] measured their type map overhead to be 57% of the size of the bytecode, compared with our 105% average (see Table 2). They chose a concise representation for the type information, which they do not compress, whereas we have chosen a wide, programmer-oriented representation and relied on compression to compensate for the wasted space. Also, they compute maps for all methods at class load time, while our scheme limits map generation to methods caught on the stack at GC time. When all methods are considered, the map space versus bytecode size ratio is lower (70%; see Table 3). We found that methods encountered on the stack at GC time tend to have more invocations than normal, a property which increases both their number of maps as well as their likelyhood to be found on the stack. Finally, the solution of Agesen *et al.* to the `jsr` problem is based on variable splitting and rewriting of the bytecode. The way we handle `jsrs` does not affect the bytecode, but does increase the size of the maps.

Stichnot *et al.* [39] describe a system totally dependent on JIT compilation where a map can be generated at any instruction. They use a two-level scheme: first, the data itself is encoded efficiently using domain-specific knowledge. Subsequently, Huffman compression is applied to the encoded data, also using compression parameters that are collected during offline training runs. They report 20%–30% overhead when compared with the size of the compiled code. Their approach is quite different from the one presented here, making a direct comparison difficult, but for reasons explained above (Section 4.4), compression rates improve as more methods are JITted. Also, Stichnot *et al.* do not compress maps separately, so the time to extract a map using their scheme could be larger.

5 Correctness of Type Maps

The generation of stack frame type maps presents a problem for testers. The map generation algorithm is far from trivial and any implementation of it requires careful testing and debugging. On the other hand, testing is not easy. Few of the maps generated are actually used under normal circumstances. Moreover, even an incorrect map is surprisingly unlikely to cause a program to fail.

Map errors could either cause retention of garbage, by identifying a slot as a reference when it is not, or allow the collection of a live object, by failing to identify a slot with a reference. Errors of the former type do not cause program to fail. Those of the latter type can do so only when the missed reference is the sole reference to its object or the referenced object has moved, and even then it is very dependent on program behavior if a noticeable failure actually occurs. This was born out by the tests described below: they uncovered rather fundamental errors in the implementation of the map generation algorithm, even after it had passed without error a variety of benchmarks and a battery of specifically designed test programs.

The test strategy consisted of three components:

1. The Jvm was instrumented to allow for direct verification of map content, independent of whether any map errors could cause program failures.
2. An automated tool for generating test program was employed to cover the more difficult aspects of map generation.
3. A tool measuring the extent to which the map generation code had been *covered* by the tests was used. Additional test programs were handcrafted to expand the coverage to those parts of the code left unexercised.

5.1 Jvm Instrumentation

In order to verify maps directly, the Jvm interpreter was instrumented to place tags on all stack slots, indicating which of them contain object references. (Such tags could in principle solve the type accuracy problem in general, but their maintenance at runtime is too time-consuming. They are therefore useful only in tests where performance is not an issue.)

The tags are not kept on the stack itself but in a separate data structure which parallels it, the *shadow stack*. Tag information is initialized upon entry to a method with the types of the arguments (kept in local variables). It is updated after the execution of any bytecode that affects the stack according to the semantics of the executed bytecode.

Whenever the execution of a method reaches a point for which a map exists, the map is retrieved from the repository and compared with the tag information for the frame. Two kinds of disagreement could occur:

1. the map indicates a reference in a slot tagged as a non-reference, and
2. the map indicates a non-reference in a slot tagged as a reference.

The first kind is clearly an error and is flagged immediately, but the second kind might not be: it is possible that the slot in question is holding some layover value that no execution path will subsequently use. Although the map generator does not perform full liveness analysis, it can detect cases where conflicting types get stored in the same slot on different execution paths. Such slots are declared unusable until they are reinitialized; code which indeed never uses them can pass verification. The instrumentation code, therefore, postpones flagging an error on this kind of mismatch until the slot is actually used. It is interesting to note that although unusable slots among the local variables are expressly permitted [29, p. 146], the verifier rule for merging operand stacks in the JVM Specification [ibid] seems to preclude them from the operand stack. In reality, however, code in which unusable slots occur on the operand stack do pass verification. (The only bytecode which can legally apply to an unusable operand stack slot is `pop`, so this does not seem like a security loophole.)

5.2 Test Programs Generation

Standard benchmarks, the JCK 1.3, and some handcrafted tests were used for testing the instrumented Jvm. However, due to the complexity of handling `jsr` bytecode in the course of map generation, test programs that use `jsr` are particularly important.

The Java compiler uses the `jsr` bytecode predictably when compiling a `try {body} finally {handler}` construct (and also for `synchronized` blocks). It is therefore possible to use test programs in Java instead of resorting to programs generated directly in bytecode.

A test-program generation tool, GOTCHA-TCBeans [22] was employed for this purpose. To use this tool, the desired behavior of the code to be tested must first be modeled using a special modeling language. The tool then analyzes the model and generates tests in an abstract language, which is later converted to actual test code; in this case, Java classes.

The model built in this case is based on the specification of the `try` statement in the Java Language Specification[20, Section 14.18, pp. 290–294]. The model includes variables, such as whether to use `catch`, `finally` or both, the nesting depth of `finally` blocks (and consequently of `jsr` subroutines), and the type of exceptions generated: intentional (caused by a `throw`), or accidental (e.g., divide by zero). GOTCHA subsequently creates tests for every permissible combination of the variables. In our case, 5783 test programs were generated.

As discussed in Section 3.1, however, many interesting situations involving `jsr` subroutines may never be generated by a compiler. To cover these cases, the test suite was augmented with programs written directly in bytecode, using a bytecode assembler/disassembler [21].

5.3 Measuring Test Coverage

To measure the extent to which the map generating code is exercised by the test programs, the code-coverage tool ATAC [7] was used. By using an ATAC-provided C compiler, which saves compile-time information and instruments the tested program to produce runtime statistics, one can determine the lines of code that have not been exercised.

ATAC provides several types of coverage statistics, but the results below pertain only to statement coverage, which regards any line of code that has been executed at least once as covered.

The ATAC-provided visualization tool can also prioritize lines of code considered uncovered according to how many additional lines are dependent on them: lines whose execution will cause the greatest number of additional lines to be covered are ranked the highest. A human test writer can then concentrate his or her effort where it is likely to make the greatest difference. The existence of unusable slots on the OPS (see Section 5.1 above) was discovered by one of the tests that was generated specifically to improve the coverage results.

5.4 Testing Results

Coverage was measured for the map generation code only, which consists of 6914 lines of code making 2649 basic blocks.

The generation of maps was exercised using tests from the following sources:

- JCK 1.3
- SPECjvm98
- Tests automatically generated by GOTCHA
- `jsr` tests, handcrafted in bytecode
- Tests in Java and bytecode, manually generated based on ATAC recommendations.

The coverage results are summarized in Table 5. As expected, the greatest contribution comes from the JCK. It is interesting to note that some JCK tests are so small that they contain no maps at all and do not contribute to the coverage. It is not surprising to see that the SPECjvm98 test suite does not add any coverage beyond what is provided by the JCK. The effectiveness of the hints ATAC provides the test writer are also evident from the data.

The 83% represents close to complete statement coverage. Most of the code in the remaining 17% is in one of the following categories:

1. Error handling code to handle situations that should never occur.

test suite	coverage added	coverage total
JCK 1.3		78.4%
SPECjvm98	0.0%	78.4%
GOTCHA	0.7%	79.1%
`jsr` tests	0.3%	79.4%
ATAC-directed	3.4%	82.8%

Table 5: Code-coverage test results.

2. Trivial transformations for quick bytecodes [28, Chapter 9] not exercised due to the testing methodology. The instrumented Jvm generates a method's maps when it is first invoked before any of its bytecodes have been converted to their quick equivalents.

3. Separate map generation for small and large methods. A small method is one with fewer than 16 slots of local variable and operand stack entries combined. Every possible bytecode is treated in either case, but the `wide` bytecode variants never occur in short methods.

6 Performance

Our implementation of mostly accurate stack scanning was compared with conservative stack scanning on an IBM prototype of J2SE v1.3 for Windows [25]. The garbage collector was essentially the same in both cases: a stop-the-world parallel mark and sweep collector, which compacts only when necessary. The only differences between the two collectors were minor changes to compaction designed to take advantage of accurately scanned stack slots. We compared both the performance of the Jvm and the efficiency of the GC. All tests were run on IBM Netfinity 7000, a PC server with 4 550MHz Pentium III Xeon^{TM} processors and 2GB of RAM running Windows NT 4.0. The programs used were taken from SPECjvm98 [38], a suite of client benchmarks, and SPECjbb2000 [37], a multithreaded server benchmark.

Table 6 presents performance data collected while running SPECjvm with a 12MB heap. We ran the Jvm in two modes:

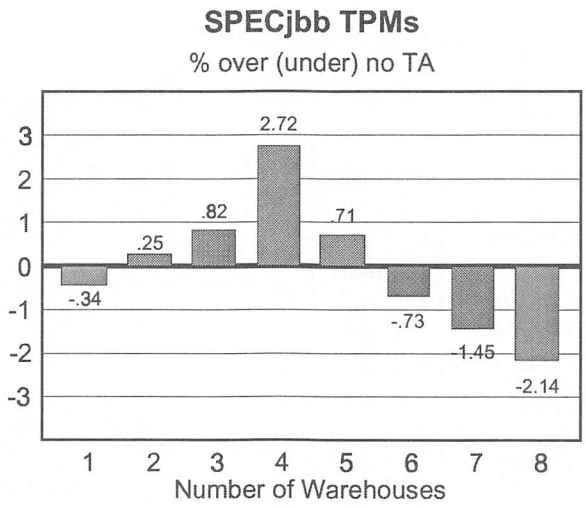


Figure 2: SPECjbb 2000 performance with TA compared with results without TA.

- with interpretation only, and
- in mixed mode, where methods are interpreted initially, and are JIT-compiled only if they are invoked sufficiently often.

The best of 10 running times in both cases was selected. The results corroborate our expectation that the overhead incurred for mostly accurate stack scanning is small.

Tables 7, 8 and 9 show how much memory is found in conservatively referenced objects. Tables 7 and 8 refer to SPECjvm using 12MB and 24MB heaps, respectively, while Table 9 describes SPECjbb running in a heap of 512MB. The tables provide object counts as well as total object sizes. The numbers represent averages of measurements taken at the end of each GC cycle. The results indicate that mostly accurate stack scanning removes a significant part of the conservatively referenced memory. In particular, the number of non-relocatable objects was reduced by 37% to 94% and the space they occupy was reduced by 40% to 99.7%. The reduction was particularly noticeable in benchmarks with the largest number of such objects. Moreover, most of the varability in test results is due to variations in the behavior of the non-TA Jvm; the number and size of immovable objects with mostly accurate stack scanning seems to be independent of heap size and number of GC cycles in the run.

SPECjbb is a benchmark which simulates a 3-tier transaction processor. Unlike other benchmarks, SPECjbb lets the application reach a steady state and then measures the rate at which its simulated server responds to its simulated clients, reported as transactions per minute (TPMs). This form of measurement makes it difficult to separate the overhead incurred from the benefits obtained when mostly accurate stack scanning is applied. Figure 2 shows the percent performance change of mostly accurate scanning with respect to the base, conservative scanner. Performance is improved up to a peek of 2.7% over base (at 4 warehouses), but at larger numbers of warehouses performance deteriorates until it drops 2.1% below base. From studying the GC trace output (not shown) it seems that although scanning the stacks accurately takes about 80% longer than it does when done conservatively, the overall time spent doing GC is practically not affected. The difference in the TPM figures must stem from other sources, currently under investigation.

7 Conclusions and Future Work

We have tried to make a case for mostly accurate stack scanning as a viable alternative to full accuracy in cases where performance outweighs GC efficiency, and also where the complexity involved in implementing full accuracy is not justified. We have shown that a mostly accurate Jvm can reduce the amount of conservatively referenced memory to very

test name	without JIT			with JIT		
	w/o TA	w/ TA	change	w/o TA	w/ TA	change
compress	291.4	288.5	-1.01%	15.6	15.6	0.0%
jess	77.7	77.3	-0.52%	13.9	13.9	-0.0%
db	158.6	155.6	-1.93%	29.3	29.2	-0.3%
javac	108.1	106.9	-1.12%	38.4	38.7	0.8%
jack	64.9	64.5	-0.62%	22.6	22.5	-0.4%
mpeg	257.1	257.2	0.04%	12.6	12.5	-0.8%
mtrt	80.2	80.2	-0.07%	8.8	8.8	0.1%
total	1038.0	1030.2	-0.76%	141.2	141.2	-0.0%

Table 6: Running times of SPECjvm98 tests with and without mostly accurate stack scanning. Tests were run with a fixed heap size of 12MB.

test name	GC cycles w/o TA	GC cycles w/ TA	Average object count total	Average object count immovable w/o TA	Average object count immovable w/ TA	Average object count immovable change	Average memory size (bytes) total	Average memory size (bytes) immovable w/o TA	Average memory size (bytes) immovable w/ TA	Average memory size (bytes) immovable change
compress	20	20	5715	51.7	26.0	49.7%	3413692	1880	880	53.2%
jess	35	35	25532	345.6	25.3	92.7%	1553343	21930	862	96.1%
db	28	28	275351	69.2	25.1	63.7%	8399507	78767	858	98.9%
javac	97	84	238768	224.3	40.5	81.9%	8632638	15079	1227	91.9%
jack	18	18	11244	82.6	26.0	68.5%	859647	3290	880	73.3%
mpeg	2	2	6217	41.0	26.0	36.6%	590608	1472	880	40.2%
mtrt	26	26	253945	199.8	28.4	85.8%	7151880	7385	1029	86.1%

Table 7: Amount of memory in conservatively referenced objects. Tests were run with a fixed heap size of 12MB. The numbers reflect the average of measurements made at the end of each GC cycle.

test name	GC cycles w/o TA	GC cycles w/ TA	Average object count total	Average object count immovable w/o TA	Average object count immovable w/ TA	Average object count immovable change	Average memory size (bytes) total	Average memory size (bytes) immovable w/o TA	Average memory size (bytes) immovable w/ TA	Average memory size (bytes) immovable change
compress	7	7	5532	50.3	26.0	48.3%	2129904	332428	880	99.7%
jess	16	16	24045	201.1	25.3	87.4%	1470944	11883	863	92.7%
db	8	8	223300	53.5	25.3	52.7%	6864361	63524	862	98.6%
javac	26	26	196537	178.7	40.4	77.4%	7194786	9189	1225	86.7%
jack	10	10	14392	75.1	26.0	65.4%	980317	3251	880	72.9%
mpeg	2	2	6189	41.0	26.0	36.6%	582660	1472	880	40.2%
mtrt	9	9	210157	130.2	27.9	78.6%	5989970	4600	1000	78.3%

Table 8: Amount of memory in conservatively referenced objects. Tests were run with a fixed heap size of 24MB. The numbers reflect the average of measurements made at the end of each GC cycle.

test name	GC cycles w/o TA	GC cycles w/ TA	Average object count total	Average object count immovable w/o TA	Average object count immovable w/ TA	Average object count immovable change	Average memory size (bytes) total	Average memory size (bytes) immovable w/o TA	Average memory size (bytes) immovable w/ TA	Average memory size (bytes) immovable change
SPECjbb	126	127	727379	268	17	93.7%	83798673	17236	859	95.0%

Table 9: Amount of memory in conservatively referenced objects used by SPECjbb. Tests were run with a fixed heap size of 512MB. The numbers reflect the average of 3 runs.

low levels while making a very small impact on performance. There is room to explore other frame types that could also be accurately scanned without hurting performance. A better understanding of the characteristics of conservative and mostly accurate stack scanning and the tradeoffs between them could provide leads to additional improvements.

We have consciously chosen a programmer-friendly format for the type maps, in which the data is redundantly encoded. A more concise representation could improve repository compression rates, but at the same time increase the overhead involved in uncompressing maps at GC time.

We have applied a comprehensive testing methodology to map generation for interpreted bytecodes. It would be interesting to extend this methodology to map generation for compiled (JITted) code. In particular, extending the shadow stack mechanism would not be trivial.

Acknowledgments: We would like to thank the many people who have contributed to this project: Alain Azagury, Victor Leikehman, Yossi Levanoni, Erez Petrank, Dafna Sheinwald, Boaz Shmueli and Sagi Snir of HRL, Toshio Nakatani, Toshio Suganuma, Tamiya Onodera, Takeshi Ogassawara and many others at TRL, Kean Kuiper of Austin Java Performance Group, and Dionis Hristov of Toronto. Special thanks go to our colleagues at the JTC: Martin Trotter, whose proposal of a shadow stack to test the maps with is but one of his many contributions to this effort, and to Sam Borman, who together have nurtured this project from its inception.

References

[1] Ali-Reza Adl-Tabatai, Michal Cierniak, Guei-Yuan Leuh, Vihesh M. Parikh, and James M. Stichnoth. Fast effective code generation in a Just-In-Time Java compiler. In PLDI [34].

[2] Ole Agesen. GC points in a threaded environment. Technical report, Sun Microsystems, Inc., 1999. TR-98-70.

[3] Ole Agesen and David Detlefs. Finding references in Java stacks. In Peter Dickman and Paul R. Wilson, editors, *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, October 1997.

[4] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java Virtual Machines. In PLDI [34], pages 269–279.

[5] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1):211–238, 2000.

[6] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. Mergen, J. C. Shepherd, and S. E. Smith. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '99)*, volume 34, 1999.

[7] ATAC: Automatic test analysis for C, 1998. Available at http://xsuds.argreenhouse.com.

[8] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, CA, February 1988. Also in Lisp Pointers 1, 6 (April–June 1988), 2–12.

[9] Joel F. Bartlett. Mostly-Copying garbage collection picks up generations and C++. Technical note, DEC Western Research Laboratory, Palo Alto, CA, October 1989. Sources available in ftp://gatekeeper.dec.com/pub/DEC/CCgc.

[10] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[11] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[12] Java Virtual Machine by Sun Microsystems, Inc. Available at http://www.sun.com/-software/sales.

[13] Robert Dimpsey, Rajiv Arora, and Kean Kuiper. Java server performance: A case study of building efficient, scalable Jvms. *IBM System Journal*, 39(1):151–174, 2000.

[14] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of SIGPLAN'92 Conference on Programming Languages Design and Implementation*, volume 27 of *ACM SIGPLAN Notices*, pages 273–282, San Francisco, CA, June 1992. ACM Press.

[15] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1994.

[16] Tamar Domany, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, June 2000. ACM Press.

[17] EVM: an exact Java Virtual Machine by Sun Microsystems, Inc. Available (as part of J2SE) at http://www.sun.com/solaris/java.

[18] S.W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3):399–401, July 1966.

[19] James Gosling. Java intermediate bytecodes. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*, pages 111–118, January 1995.

[20] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997.

[21] Matt Greenwood and Sara Porat. TOAD - an environment for monitoring understanding and optimizing Java. In *OOPSLA '99 Companion*, 1999. Available at http://www.alphaworks.ibm.com/tech/toad.

[22] Alan Hartman and Kenneth Nagin. TCBeans Software Test Toolkit. In *Proceedings of the 12th International Software Quality Week*, May 1999. Available at http://www.haifa.il.ibm.com/projects/gtcb.

[23] Java HotSpot Technology by Sun Microsystems, Inc. Available at http://java.sun.com/-products/hotspot.

[24] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

[25] IBM 1.3 Java Developer Kit for Windows. Available at http://www.software.ibm.com/-download.

[26] Open runtime platform by Intel, Inc. Available at http://intel.com/research/mrl/orp.

[27] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[28] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1st edition edition, 1997.

[29] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 2nd edition edition, 1999.

[30] SUN Microsystems. Java Native Interface Specification. Technical report, Sun Microsystems, Inc., http://java.sun.com/j2se/-1.3/docs/guide/jni/, 1997.

[31] SUN Microsystems. JNI enhancements in JDK 1.2. Technical report, Sun Microsystems, Inc., http://java.sun.com/j2se/-1.3/docs/guide/jni/jni-12.html, 1998.

[32] David A. Moon. Garbage collection in a large LISP system. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–245, Austin, TX, August 1984. ACM Press.

[33] Tamiya Onodera. A generational and conservative copying collector for hybrid object-oriented languages. *Software Practice and Experience*, 23(10):1077–1093, October 1993.

[34] *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.

[35] Tony Printezis and David Detlefs. A generational mostly concurrent garbage collector. In *Proceedings of the International Symposium on Memory Management (ISMM00)*, pages 143–154, Minneapolis, MN, October 2000.

[36] Jacob Seligmann and Steffen Grarup. Incremental mature garbage collection using the train algorithm. In O. Nierstras, editor, *Proceedings of 1995 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, University of Aarhus, August 1995. Springer-Verlag.

[37] SPECjbb2000 Java Business Benchmark. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1998. Available at http://www.spec.org/osg/jbb2000/.

[38] SPECjvm98 Benchmarks. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1998. Available at http://www.spec.org/osg/jvm98/.

[39] James M. Stichnoth, Guei-Yuan Lueh, and Michal Cierniak. Support for garbage collection at every instruction in a Java compiler. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 118–127, Atlanta, May 1999. ACM Press.

[40] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

Hot-Swapping between a Mark&Sweep and a Mark&Compact Garbage Collector in a Generational Environment

Tony Printezis

Department of Computing Science, University of Glasgow,
17 Lilybank Gardens, Glasgow G12 8RZ, Scotland.

tony@dcs.gla.ac.uk

Abstract

This paper describes a novel method for dynamically switching between a Mark&Compact (M&C) and a Mark&Sweep (M&S) garbage collector in the generational memory system of a high performance Java virtual machine. A M&C collector reclaims space by sliding all live objects towards the beginning of the heap. A M&S collector de-allocates garbage objects in-place. In this paper, both algorithms are assumed to operate over the old generation of a generational memory system and on their own provide different trade-offs to the application that uses them: faster old collections but with slower young collections and the possibility of fragmentation (M&S) or slower old collections but with faster young collections and the guarantee to eliminate fragmentation (M&C). We propose *Hot-Swapping*, a technique for dynamically switching between these two algorithms, to attempt to achieve the "best of both worlds". Its introduction to the memory system of the virtual machine imposed minimal changes to the existing implementations of M&C and M&S and virtually no extra performance overhead. Experimental results, presented in the paper, show that this hybrid scheme can either outperform both algorithms, or is very close to the faster of the two (whether this is M&S or M&C), while never being the slowest.

1 Introduction

Programming languages that rely on garbage collection [14] for their memory management have existed since the late 1950s. Even though the advantages of garbage collection (ease-of-development, program robustness, etc.) are well-known and accepted, most software developers have continued to rely on the traditional explicit memory management (`malloc/free` in C, `new/delete` in C++, etc.), largely because of performance concerns. Recently, however, the wide acceptance of the Java™ programming language [12] has encouraged developers to take advantage of the benefits of garbage collection and has stirred further interest in the area.

Generational garbage collection techniques [17, 25] can address the performance concerns usually associated with automatic memory management. They divide the heap into spaces, referred to as *generations*, according to object age. Since, for most programs, young objects are more likely to be garbage than older ones, concentrating collection activity on the young space increases throughput, as more free space is reclaimed per collection cycle. The young space is kept small to allow fast non-disruptive collection times.

Objects that survive a given number of young collections are considered long-lived and are *promoted* into an older generation. Even though older generations are large, they are not of infinite size, therefore they will eventually run out of space and require collection. Two well-known and widely-used algorithms that can perform the collection are Mark&Sweep and Mark&Compact [14, 27]. The former reclaims space by de-allocating it in-place and keeping track of it in order to re-use it later. The latter reclaims space by sliding (compacting) all live objects towards the beginning of the space, thus creating a single contiguous free chunk.

The two garbage collection algorithms described above have different performance characteristics and each one will perform best for specific loads. Mark&Compact provides fast allocation to old space, hence providing faster young collection times, and eliminates *fragmentation* [13]. Mark&Sweep provides faster old collection times, but can suffer from fragmentation and slower young collection times.

This paper proposes a technique to dynamically switch (*hot-swap*) between Mark&Compact and Mark&Sweep in order to achieve the "best of both worlds". This technique is simple to implement and imposes virtually no performance overhead on the memory system. Experimental results, presented in the paper, show that it can

either outperform both garbage collection algorithms on their own, or be very close to the faster one (whichever that is), while never being the slowest.

The idea of dynamically switching between different garbage collection algorithms has been successfully adopted before [7, 21, 16]. However, the novelty of the technique proposed here lies in the fact that it uses two collection algorithms that provide, in turn, faster old collections/slower young collections and slower old collections/faster young collections and tries to find a winning balance between them. As far as the author is aware, this aspect of the work has not been explored before.

It must also be noted that this paper does not address incremental garbage collection. Instead, it presents a technique that can improve the performance of batch jobs (compilation, raytracing, etc.), whose main requirement is throughput and not non-disruptive garbage collection pauses.

A longer version of this paper, which contains more graphs generated from the experimental results, is also available as a technical report [18].

1.1 The Implementation Platform

The Sun Microsystems Laboratories Virtual Machine for Research, henceforth *ResearchVM*, which was used as the implementation platform for all experiments presented in this paper, is a high performance Java virtual machine developed by Sun Microsystems. This virtual machine has been previously known as the "Exact VM", and has been incorporated into products; for example, the Java 2 SDK (1.2.1_05) Production Release, for the Solaris™ operating system. It employs an optimising just-in-time compiler [10] and a fast object-synchronisation mechanism [3].

More relevantly, it features high-performance *exact* (i.e., non-*conservative* [8], also called *precise*) memory management [2]. The memory system is separated from the rest of the virtual machine by a well-defined *GC Interface* [26]. This interface allows different garbage collectors to be "plugged in" without requiring changes to the rest of the system. A variety of collectors implementing this interface have been built. Above the GC Interface, a second layer called the *generational framework* facilitates the implementation of generational garbage collectors.

1.2 Terminology

Some terminology and abbreviations that will be used throughout the paper are enumerated below.

- *Mark&Sweep* (M&S): a stop-the-world GC that reclaims free space by de-allocating garbage objects in-place [14].
- *Mark&Compact* (M&C): a stop-the-world GC that reclaims space by sliding (compacting) live objects towards the beginning of the heap, providing a single contiguous area of free space [14].
- *Hot-Swapping* (H-S): the mechanism to dynamically switch between a M&S and a M&C GC, described in this paper.
- *Young GC*: a young generation GC in a generational memory system [25, 14].
- *Old GC*: an old generation GC in a generational memory system [25, 14].

1.3 Paper Overview

Section 2 provides the motivation behind this work by comparing and analysing the M&C and M&S algorithms and presenting the different trade-offs each of them provides. Section 3 outlines the Hot-Swapping mechanism, which we propose in order to achieve the best balance between the two algorithms mentioned previously. Section 4 presents and analyses timing results from the experiments. Section 5 covers the related work and section 6 concludes the paper.

2 Motivation

During our work on a different project (the *Generational Mostly-Concurrent GC*, discussed elsewhere [19, 20]), we needed to implement an in-place de-allocation mechanism to be applied to the old generation of the memory system of ResearchVM. However, we decided to first implement, test, and optimise this mechanism in the context of a non-concurrent stop-the-world M&S GC, for simplicity and predictability reasons.

When we compared the performance of the M&S GC and that of the default M&C GC (both of them were applied to the old generation of the system and shared the same semispace-based young generation), we made the following observations.

- M&S could decrease the maximum old GC pause times by a factor of 2 or 3, compared with M&C.
- Unfortunately, and somehow surprisingly, M&S also imposed a performance penalty of up to a factor of 2 on *young* GC pause times (↝§2.1 and §2.2).
- M&S sometimes required a larger heap, compared with M&C, in order to operate. This was mainly due to fragmentation [28, 13].

Given the above, it is clear that an application which performs a lot of frequent heavy-duty old GCs could benefit from using M&S, whereas one that mainly performs young GCs could benefit from using M&C, which also has the advantage of eliminating fragmentation. The motivation for the work presented in this paper follows naturally: whether it is possible to dynamically switch between a M&S and a M&C old GC and achieve beneficial performance results by getting the "best of both worlds".

It should be noted that the applications we have observed do not seem to radically change their behaviour during their execution. The hybrid scheme described here is not proposed solely as a solution to detect and deal with application behavioural changes, but also to benefit applications that have uniform behaviour.

The remainder of this section will compare the M&C and M&S algorithms and will demonstrate how they affect the performance of our generational memory system. Section 2.1 covers object-allocation-related issues. Section 2.2 overviews the young GC mechanism and section 2.3 covers old GC issues. Then, section 2.4 presents measurements to illustrate the performance differences between the two algorithms.

2.1 Allocation Issues

As described in section 1.2, the main difference between the M&S and M&C GCs is their allocation and de-allocation policies.

During GC, M&S de-allocates garbage objects in-place. It keeps track of where free space is located within the heap and uses this information to satisfy future allocation requests.

The particular technique that we use to keep track of free chunks in our implementation is *free lists, segregated by size*. We maintain separate free lists for free chunk sizes up to 128 4-byte words, segregated free lists for sizes up to 32K words, and a single large-free-chunk list for all larger sizes. During each old GC cycle, the free lists are re-created from scratch, rather than updated. Additionally, all adjacent free chunks are coalesced into single bigger ones.

It is worth pointing out that, according to Johnstone and Wilson, the use of segregated free lists is one of the worst allocation policies for generating fragmentation [13]. Our decision to choose it had originally been based solely on the grounds of simplicity. However, for the purposes of the hybrid mechanism described and analysed in this paper, fragmentation generated by M&S is not a major issue, as M&C will eventually eliminate it. Hence, we did not think that it was necessary to explore alternative allocation policies.

Considering M&C, this GC slides all live objects towards the beginning of the space, thus generating a single contiguous free chunk. Allocation off this free chunk can be very fast, using the well-known "bump a pointer and check" technique. This is clearly faster than any alternative technique (in particular, allocation from free-lists) and gives M&C an allocation performance advantage over M&S.

We consider the two algorithms when applied to the old generation of a generational memory system. In such a system, most allocations to the old generation take place in bursts during young GCs, as objects are evacuated from the young generation[1]. This observation provides an opportunity to optimise the allocation operation of M&S in the following manner: if a very large free chunk can be discovered, then linear allocation can be performed inside it, eliminating the need for look-ups on the free lists. This optimisation is very effective early in an application's execution and allows M&S to allocate objects almost as efficiently as M&C. However, an appropriately large free chunk is not always available, especially after a few old GC cycles when the space starts to become fragmented, and M&S typically has to eventually revert to allocating directly from the free lists.

It is worth pointing out that, no matter how the old generation is managed, the front-line allocator that all applications will use directly is a fast "bump a pointer and check", provided by the semispace-based young generation (see footnote 1 for an exception to this rule). So, an old generation with a slower allocator (e.g. M&S) will not affect the performance of most allocations.

To summarise, the speed of allocation to the old generation directly affects the performance of young GCs. As M&C has an inherently faster allocation operation, it also has the potential to support faster young GCs.

2.2 Young Generation GC Operation

Figure 2 illustrates the operation of young GCs over M&C. Objects from the young generation are promoted to the old generation using a Cheney-style copying technique [9]. Objects reachable from the old generation are promoted first. Forwarding pointers are installed in their old image in order to specify that an object has already been copied, to avoid copying it again if it is referenced multiple times. Then, the reference fields of newly-promoted objects are visited and, if they point to young objects, those objects are promoted too. This process is repeated until a given threshold of young objects has been promoted (this is adjusted dynamically).

The allocation operation of M&C (⇝§2.1) guarantees

[1] The main exception to this rule is very large objects that do not fit in the young generation and have to be allocated directly in the old one. This, however, is relatively infrequent.

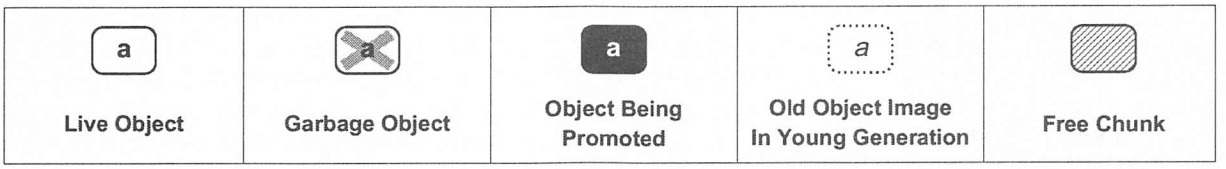


Figure 1: Legend for figures 2, 3, 7, and 9.

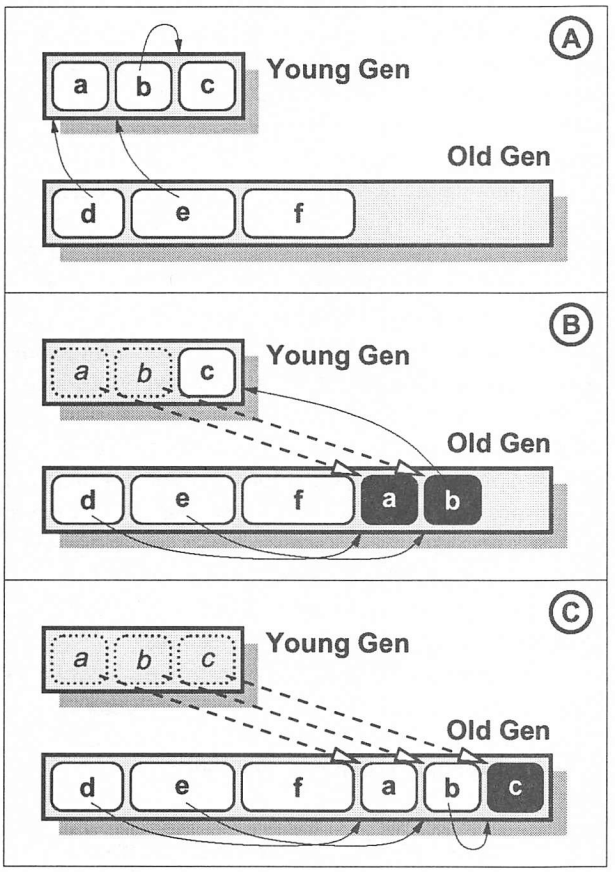


Figure 2: Young GC Operation for M&C.

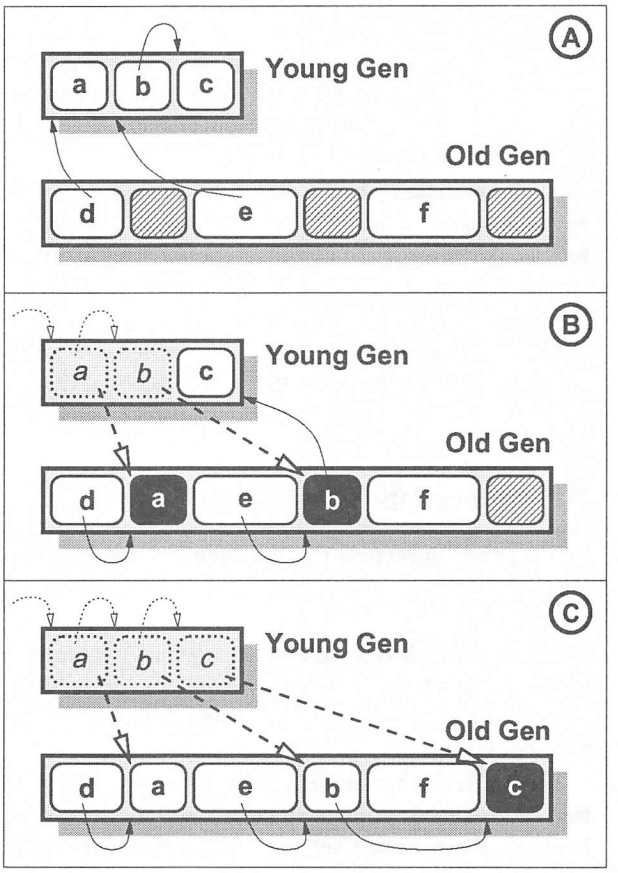


Figure 3: Young GC Operation for M&S.

that all newly-promoted objects will be allocated linearly into the old generation. This allows the above algorithm to discover them and iterate over them (in order to potentially promote more objects) very efficiently.

Figure 3 illustrates the operation of young GCs over M&S. This is largely similar to the one corresponding to M&C, with one important difference: M&S does not guarantee that all newly-promoted objects will be allocated linearly into the old generation. This makes scanning them less efficient, compared with M&C.

The way we achieve this is to chain the old images of newly-promoted objects into a linked list and iterate over it when we need to iterate over promoted objects. The forwarding references from the old images of the objects to the newly-allocated ones need to be installed anyway (for the reasons described above), hence the only extra expense is the construction of the list. In practice however it turns out that this is not as efficient as the equivalent operation of M&C.

To summarise, M&C not only provides faster allocation to the old generation (⇝§2.1), it also has the potential to allow faster object-promotion operations (e.g. iteration over newly-promoted objects) during young GCs. These two advantages that M&C has over M&S are, we believe, the main causes of the faster young GC times that can be achieved using M&C (timing results that demonstrate this

are included in sections 2.4 and 4).

2.3 Old Generation GC Issues

Even though M&C outperforms M&S in young GC times (⇝§2.1 and §2.2), it performs worse in old GC times, even up to 2–3 times slower than M&S. The main reason for this is that the compacting phase of M&C (in our implementation, a variation of the *Lisp 2 Algorithm*, described in Jones' book [14]) needs to

- relocate objects, possibly causing excessive memory copying, and
- patch reference fields to reflect the object relocations.

These two operations turn out to be more expensive than M&S's relatively simple operation of adding free chunks to free lists. However, they can affect performance only if a non-trivial percentage of garbage objects resides in the heap. If there are no garbage objects, no live objects need to be relocated and no reference fields need patching. In practice, if the percentage of garbage objects is very low, old GC times for M&S and M&C are very close. However, as this percentage increases, M&S starts outperforming M&C.

Experimental results in section 4 reflect the claims made in this section on old GC times.

2.4 Measurements

This section contains a concrete example to demonstrate the claims on young and old GC times that have been presented so far. Figure 4 shows the behaviour of young GC times for the large Javac benchmark (⇝§4.1), when using M&C. Each point represents a single young GC, the x-axis represents the start of the young GC (in seconds into the benchmark), and the y-axis represents the duration of the young GC (in milliseconds). Additionally, the two old GCs that took place are indicated with vertical arrows.

The figure shows that the behaviour of the application clearly affected the young GC times. It has in fact three distinct phases: the first one up to 21 seconds into the application (slower and more varied than the rest), the second one between 21 and 50 secs (more consistent and slightly faster than the rest, but with a few outliers), and the third one after 50 seconds (the end of the first and second phase is indicated in the figure).

Figure 5 shows the behaviour of young GC times for the same benchmark, when using M&S. Its data points, again representing single young GCs, are split into the following four categories.

① **100% Linear**: all the allocations to the old generation took place linearly to a single contiguous free chunk.

② **90%–100% Linear**: between 90% and 100% of allocations to the old generation took place linearly, the rest were directly off the free lists. For the ones that took place 100% linearly, no single contiguous free chunk that was large enough could be found, hence at least two non-contiguous ones were used.

③ **0%–90% Linear**: between 0% and 90% of allocations to the old generation took place linearly.

④ **0% Linear**: no allocations to the old generation took place linearly and all of them were directly off the free lists.

Looking at figure 5, and comparing it with figure 4, the following observations can be made.

- It is clear that young GC times using M&S are generally slower compared with the young GC times over M&C. This is most prominent during the first phase of the benchmark (0–21 seconds).
- The majority of young GCs from category ① took place before the first old GC (as the heap had not been fully used and a single contiguous free space was still available). Subsequently, after free space had been made available by the old GCs, contiguous large free chunks had not been produced, hence most young GCs did not fall into category ①. Interestingly, after each old GC, allocations still took place mostly-linearly but to smaller and more than one contiguous chunk (category ②). Then, when these were exhausted, all allocations took place off free lists (category ④). We believe that the above is clear evidence of fragmentation being introduced in the heap[2].
- Figure 5, when compared with figure 4, shows some evidence that the category ② and ④ young GCs are slightly slower than the ones of category ① (their data points seem to be higher).

Further, figure 6 shows the minimum, average, and maximum times for young and old GCs for the same benchmark. For the reasons described above, M&S does provide slower young GC times, compared with M&C. However, as seen on the same figure, it also provides much faster old GC times[3].

[2] Johnstone and Wilson claim that fragmentation only exists in a heap when the free space is fragmented *and* the free chunks are too small to satisfy individual allocation requests [13]. In this paper, we will use a slightly different definition of fragmentation: *"the heap is fragmented and no large enough free chunks are available to support linear allocations during young GCs, even if free chunks are available to satisfy non-linear allocations."*

[3] The figure is slightly misleading because, as it has already been mentioned, only two old GCs took place. Still, this result follows the general pattern we have observed with old GC times.

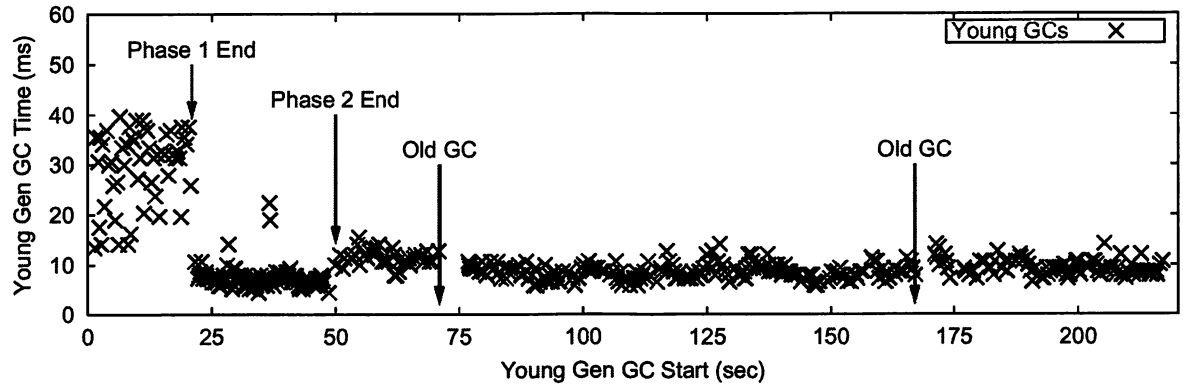


Figure 4: **Javac** benchmark with M&C — young GC trace.

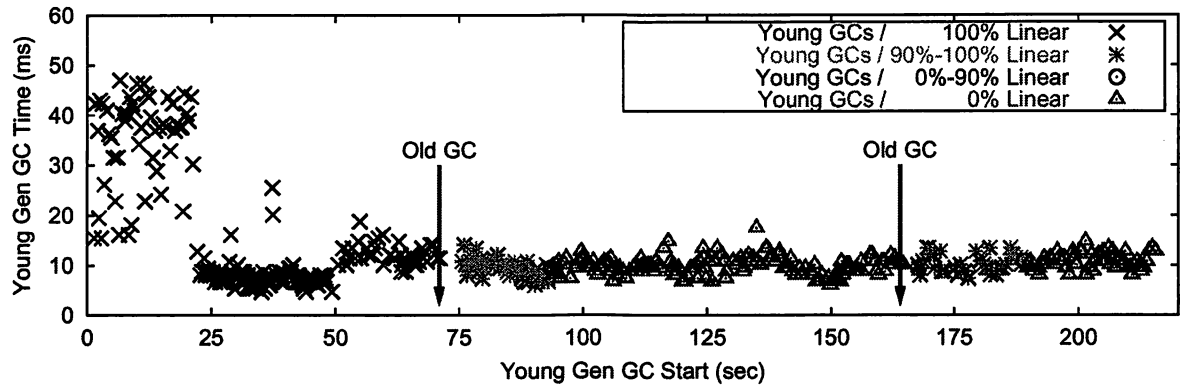


Figure 5: **Javac** benchmark with M&S — young GC trace.

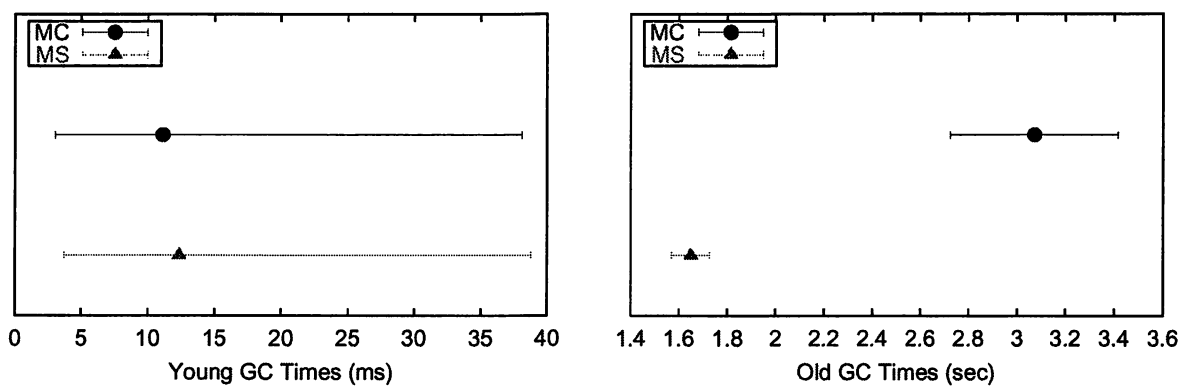


Figure 6: **Javac** benchmark, GC time distributions (min, avg, max).

From the above comparison, it would have been natural to conclude that M&S should perform better overall as, after all, its old GCs are seconds faster than those of M&C and its young GCs only a few milliseconds slower. Still, the fact that around 2,270 young GCs took place, compared with only 2 old GCs, amplified the slowness of young GCs and did not make the faster old GCs as beneficial as originally thought. It turns out that the total GC time for M&S was only 0.03 seconds faster than that of M&C.

We have observed this pattern (elapsed times being very similar for both algorithms) in a variety of applications. However, if an application performs mainly young GCs and very few old GCs, it will benefit from M&C. Alternatively, if it relies on old GCs, it will benefit from M&S. We have observed applications that fall into these two categories and measurements for some of them are presented in section 4.

3 The Hot-Swapping Mechanism

Section 2 demonstrated the differences in performance and trade-offs between M&S and M&C, when applied to the old generation of a generational memory system. This section presents a mechanism for dynamically switching between M&S and M&C, in an attempt to achieve the "best of both worlds". This mechanism will be referred to as *Hot-Swapping* (H-S). Section 3.1 presents the requirements that we imposed on H-S. Sections 3.2 and 3.3 describe how the the switch between M&C and M&S can be performed efficiently. Then, section 3.4 outlines the heuristic used to determine when to perform the switch and section 3.5 concludes.

3.1 Requirements

The requirements that we imposed on the H-S mechanism were the following.

① The switch between M&C and M&S should happen efficiently, preferably in constant time.

② Apart from the mechanisms to switch between the two algorithms and to determine whether to perform the switch, no further performance penalty should be imposed.

③ Flexibility should be provided on when the switch between M&C and M&S is allowed to be performed, so that different heuristics can be easily adopted and evaluated.

④ Only minimal changes should be imposed on the implementation of the existing M&C and M&S algorithms.

3.2 Switching from M&C to M&S

The operation to switch from M&C to M&S is straightforward. M&C ensures that all the free space resides at the end of the heap. When switching to M&S, it is only necessary to add a single free chunk that spans all the free space (this can be done in constant time). Having done this, M&S can operate over the heap as it would normally have done. Figure 7 illustrates this operation with a concrete example. First, a free chunk is added at the end of the heap. Then, M&S operates as it would normally do, finding **a**, **c**, and **f** to be garbage and replacing them with free chunks.

3.3 Switching from M&S to M&C

Switching from M&S to M&C is slightly more complicated. The reason for this is the presence of the free chunks scattered around the heap, which M&C should not be aware of. One way to deal with this, is to iterate over all free chunks and transform them to unreferenced Java objects (scalar arrays), which M&C will consider garbage and reclaim. However, there might be a large number of free chunks in the heap and these transformations might prove to be a performance overhead.

To avoid having to apply the format changes during the switch, we decided to change the format of free chunks and "mask" them as garbage objects. Figure 8 illustrates the format of scalar arrays in ResearchVM [26]. The object header of an array contains three fields: *(i)* a pointer to its NearClass[4], *(ii)* a flags field that is mainly used by the synchronisation mechanism [3], and *(iii)* a field containing the length of the array.

Figure 10 illustrates the updated format of free chunks. The first field of the free chunk header points to a NearClass constructed for this purpose (essentially a copy of the scalar array NearClass, but with a different address). The second field is used for linking free chunks in the free lists (↝§2.1). The third field contains the length of the free chunk. This format allows M&S to determine whether an object is a free chunk or not (by comparing its NearClass address to the well-known fake NearClass), determine its length, and have access to a field that can be used for linking purposes. Additionally, it allows M&C to consider free chunks as garbage scalar arrays (note that the GC does not use the flags field during its operation,

[4] A *NearClass* is a small data structure that contains GC-related information about instances of that class, mainly on object layout, type of each field, type of entries (in the case of arrays), etc. All objects point to their corresponding NearClasses and they, in turn, point to the full class structure [26]. The NearClass structure was introduce in an attempt to allow more efficient access to the information that the GC will need during its operation by keeping it grouped together, hopefully causing less secondary cache misses.

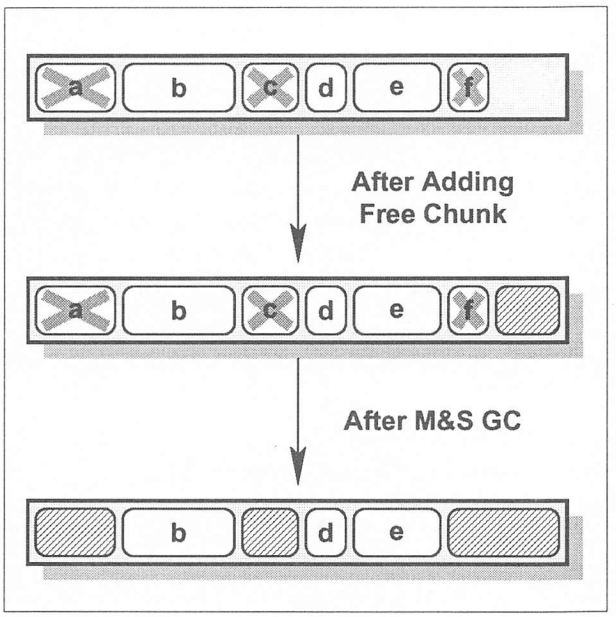


Figure 7: Switching from M&C to M&S.

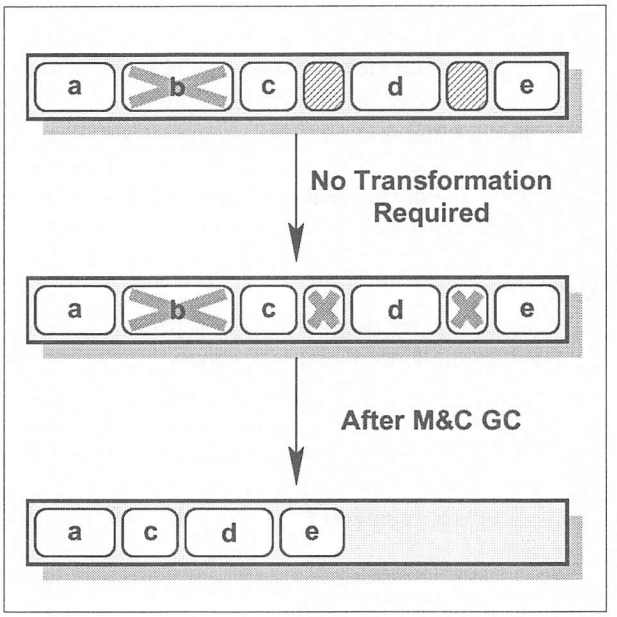


Figure 9: Switching from M&S to M&C.

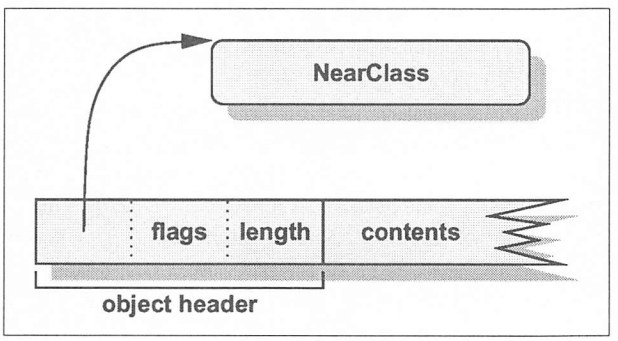


Figure 8: Scalar array format in ResearchVM.

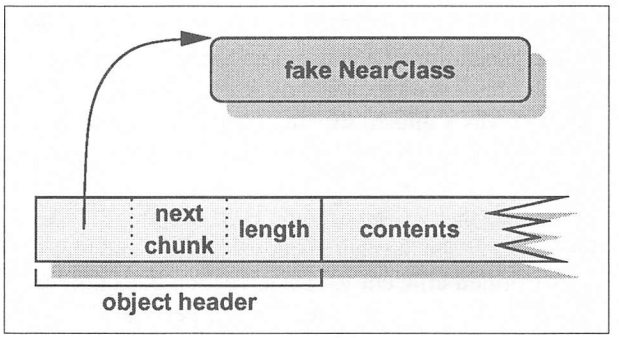


Figure 10: Free chunk "masked" as scalar array.

therefore M&C will not try to access the second field of the object header).

Given the above format of free chunks, switching from M&S to M&C is very efficient as no changes to the heap are necessary: all free chunks will be considered as garbage objects by M&C and their space will be reclaimed. Figure 9 illustrates this operation. M&C operates directly over the heap and compacts live objects **a**, **c**, **d**, and **e**, reclaiming the space taken up by the two free chunks and garbage object **b**, providing a single contiguous area of free space.

It is worth mentioning here that an alternative way to perform the switch from M&S to M&C would have been to modify M&C to be aware of free chunks. However, we chose the approach outlined above purely on grounds of simplicity, as it was less complicated to modify the format of free chunks and leave M&C totally unchanged.

3.4 Heuristic

Sections 3.2 and 3.3 demonstrated how the switch between M&C and M&S is accomplished. However, an interesting question is when to hot-swap between the two GCs. The heuristic that we implemented is the following.

> *"Use M&C after expanding the heap to take advantage of its fast allocation, otherwise use M&S to take advantage of its fast old GC times, unless linear allocation fails sufficiently, in which case switch back to M&C once to eliminate fragmentation and move back to M&S."*

A more detailed explanation follows.

- At the beginning, and also immediately after a heap expansion, a large free area is available at the end of the heap, therefore we use the operation of M&C to

satisfy allocation requests to the old generation, as it is the fastest.

- When an old GC is initiated, we use M&S (after switching from M&C if necessary), as it provides faster old GC times. Subsequently, we use the M&S allocation mechanism to satisfy allocation requests to the old generation.

- If less than 60% of young GCs since the last old GC did not use full linear allocation (i.e. less than 60% of young GCs were not in category ①, ⇝§2.4), ensure that the next old GC will switch from M&S to M&C, because the heap is assumed to be fragmented and M&C will eliminate this.

The above heuristic, even though it is simplistic, attempts to take advantage of the best qualities of each GC. It also efficient to implement, as only a flag needs to be set when linear allocation fails and two counts need to be updated once per young GC.

3.5 Summary

This section has presented a mechanism to efficiently hot-swap between a M&S and a M&C GC. It satisfies the requirements enumerated in section 3.1 because the switch between the two algorithms can be performed in constant time (requirement ①), no further restrictions (apart from the ones already in place, e.g. all the threads must be stopped [1]) are imposed on when the switch can take place (requirement ③), and the only necessary change was the alteration of the free chunk format, with the M&C code being left completely unaltered (requirement ④). Additionally, the GC Interface of ResearchVM uses virtual calls to invoke most operations implemented by each GC [26]. This allows us to, say, change the allocation operation from that of M&C to that of M&S by simply replacing the appropriate virtual call and imposing no further performance penalty. This satisfies requirement ②.

4 Measurements

This section presents the timing results from the experiments that were performed to compare the performance of H-S to that of M&C and M&S on their own. Section 4.1 gives a brief description of the six benchmarks used and section 4.2 presents and analyses the timing results.

All experiments were run on a lightly-loaded Sun Ultra™ 80 workstation [22] with four 450MHz UltraSPARC-II CPUs and 2GB of main memory running the Solaris 7 operating system. All timing results reported were obtained using the `gethrtime` Solaris call and are the average of ten runs, after the worst time has been removed.

4.1 The Benchmarks

The six benchmarks we used for our evaluation were the following.

- **GCBench**[5]: A benchmark written to stress the allocation and promotion operations of the Java system. It creates a large long-lived tree and then spawns two threads, which create smaller and shorter-lived trees. This process is repeated five times.

- **MarkTest**: A benchmark written to evaluate optimisations applied to the marking phase of old GCs. It allocates a very large object array and assigns repeatedly new objects to its entries in an attempt to generate garbage. This process is repeated ten times.

- **GCOld**: A benchmark written to evaluate the performance of incremental GCs. It creates disjoint tree structures and performs operations on them, creating short- and long-lived objects, while rendering some others garbage. It is described in more detail elsewhere [19, 20].

- **JOS**: It de-serialises a tree data structure from a file using the standard Java Object Serialization™ facilties [23], performs some updates to it, and re-serialises it to a second file.

- **DNA**: An application that creates a suffix tree [24], populates it with a DNA sequence, and performs searches over it. The process is repeated for 5 different parts of a DNA sequence (that of merged genomes of two kinds of yeast: saccharomyces cervisæ and saccharomyces pombæ).

- **Javac**: A large Javac job that compiles all the standard classes of the Java 1.2.2 distribution. It reads 2,740 `.java` files, containing 776,488 lines of code, and generates 4,638 `.class` files.

While the first four benchmarks are clearly synthetic, the last two are actually "real" applications.

4.2 Results

Figure 11 plots the total GC times (i.e. the amount of time the application was stopped for either a young or an old

[5] This benchmark was originally written by John Ellis, Pete Kovac, and Hans Boehm. We have altered it so that it has a longer elapsed time and uses more memory. Its original version is available from `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench/`.

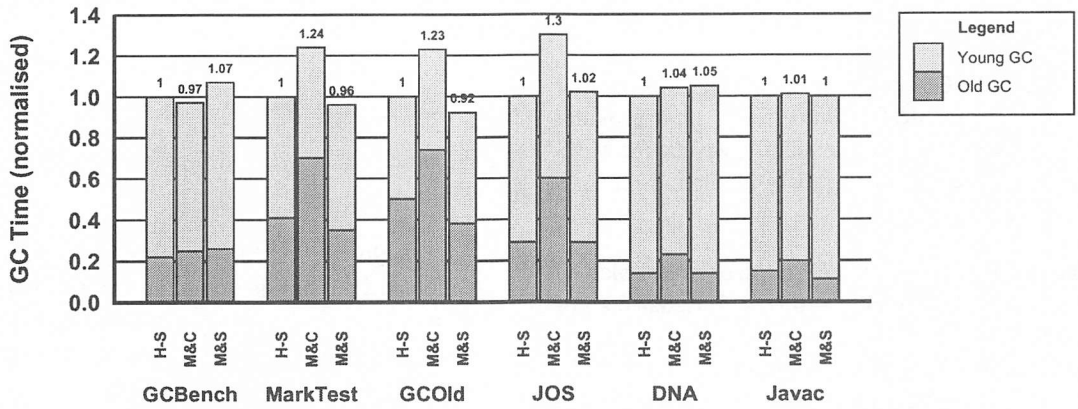


Figure 11: Young/Old GC times, normalised with respect to H-S (less is better).

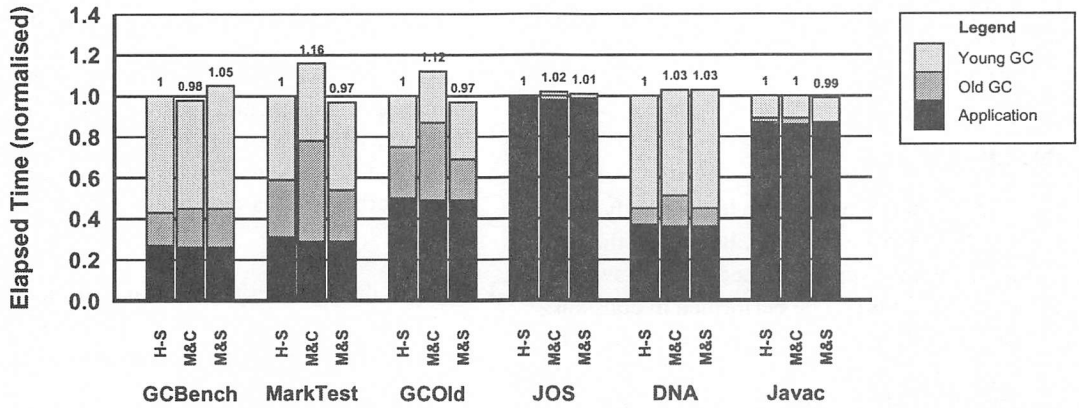


Figure 12: Elapsed times, normalised with respect to H-S (less is better).

GC) for all benchmarks, normalised with respect to H-S. The results show that, out of the three GC algorithms, no one performed best in all cases. However, H-S does appear to be the best choice as it performs best in three cases (**JOS**, **DNA**, and **Javac**) or is closer to the faster one, whether that is M&C (**GCBench**) or M&S (**MarkTest** and **GCOld**). It is interesting to note that, in the case of the **JOS** and **DNA** benchmarks, H-S outperforms the other two, even though M&C provides overall lower average young GC times and M&S lower average old GC times.

Figure 12 plots the application elapsed times, normalised with respect to H-S. This figure follows the pattern of figure 11, however the difference between the three GC algorithms is not as obvious; this is especially the case for the **JOS** and **Javac** benchmarks for which the GC times were a small fraction of the application elapsed times (interestingly, these are the two benchmarks that performed a lot of I/O).

It is important to note that, even though both H-S and M&C could run the **JOS** benchmark with a 90MB old generation, M&S failed to do so due to fragmentation-related reasons (an allocation for a large 8MB array was failing, even though there was at least twice as much space free in the generation in total). Additionally, when executing the **GCBench** benchmark, M&S performed one more old GC compared with H-S and M&C (see table 1), again due to fragmentation-related reasons. These two points are evidence to support that H-S can operate with the same heap sizes to those of M&C, even if they cause fragmentation problems to M&S.

Finally, it must be pointed out that, for a given benchmark, the difference in total GC times between two of the GC algorithms matches closely, as expected, the difference in application elapsed times. The only exception to this rule is the **Javac** benchmark. The reason behind

this is the excessive disk accesses that this benchmark performs (reading `.java` files and writing `.class` files) that caused the timing results to be slightly unpredictable (even though the total GC times obtained from that benchmark seemed to be consistent).

All the timing results obtained from the six benchmarks, when run with H-S, M&C, and M&S, are included in appendix C (tables 1–6 — entries surrounded by ||s indicate the best result out of the three algorithms).

5 Related Work

For a good introduction to garbage collection, the reader is referred to two excellent publications which touch on most of the techniques mentioned in this paper: Jones' book [14] and Wilson's survey [27]. On a related topic, Wilson, Johnstone, and others survey different dynamic allocators and discuss the problem of memory fragmentation [28, 13].

Generational garbage collection techniques were originally proposed by Lieberman and Hewitt [17], but Ungar reported the first implementation [25]. Appel has provided further analysis of their benefits [4, 5, 11].

Switching garbage collectors dynamically is not a new idea and has already been explored in the past. There are three notable efforts in this area.

Sansom proposed *dual-mode garbage collection* that switches between a single-space compacting [15] and a two-space copying [9] garbage collector. His motivation was to achieve maximum performance but improve on the high space requirements that the copying collector requires.

Bartlett's *mostly-compacting garbage collector* [6] is targetted for a system with ambiguous garbage collection roots. It allows live objects to be relocated, as long as they are not ambiguously referenced. To improve performance, a generational framework was added later [7]. The motivation behind this work was purely to take advantage of the benefits of compaction in an environment with ambiguous roots.

Finally, Lang and Dupont proposed an *incremental incrementally-compacting garbage collector* [16]. According to this algorithm, most free space is de-allocated in-place during a collection phase, with a small region of the heap being compacted using a two-space copying collector. This is performed by piggy-backing a Cheney-style copying operation [9] on the marking phase of the collector. The region to be evacuated is chosen by splitting the heap into fixed size regions and cycling through them. This scheme has similar motivation to ours (i.e. uses both in-place de-allocation and compaction, the latter to decrease fragmentation). However, it does require extra memory to operate (one region must always be free in order to evacuate live objects to) and it is not clear whether always compacting a small region of the heap during each collection is more beneficial than compacting the whole heap when necessary (provided total pause time is not an issue).

6 Conclusions and Future Work

This paper compares the performance and trade-offs of two stop-the-world garbage collectors, Mark&Compact and Mark&Sweep, when applied to the old space of a generational memory system of a high-performance Java virtual machine. It then proposes Hot-Swapping, a technique to dynamically switch between these two collectors in order to benefit from the advantages of both algorithms. Performance results, included in the paper, show that the H-S technique can either outperform M&C and M&S, or be slightly slower than the faster of the two, while never being the slowest.

We are interested in further improving the Hot-Swapping idea. We would like to determine what effect on the performance of the H-S mechanism, if any, have some of its parameters (young generation size, object-promotion order, etc.). Further, the heuristic on when to hot-swap (⇝§3.4) is currently very simplistic and we would like to implement and evaluate alternatives.

Finally, the ability of our hybrid scheme to eliminate fragmentation and improve young GC times seems very attractive for our *mostly-concurrent generational GC* [19], which currently relies only on free-list-based allocation and is sometimes affected by these two problems. Hence, we would like to explore the possibility of incorporating a form of hot-swapping in such a concurrent environment.

A Acknowledgements

This work was supported by Sun Microsystems with funding through Sun's External Research Program, that also kindly donated the workstation used for the development and evaluation of the ideas presented in this paper. The author is grateful to the members of the Java Technology Research Group at SunLabs East, especially Steve Heller, Dave Detlefs, and Alex Garthwaite, for their support, contributions, ideas, and all the food over the last few years! The author would also like to thank Huw Evans, Rolf Neugebauer, Andy King, Ole Agesen, Richard Jones, and Malcolm Atkinson for their life-saving, last-minute, constructive feedback and Ela Hunt for providing the **DNA** application.

B Trademarks

References

[1] O. Agesen. GC Points in a Threaded Environment. Technical Report TR-98-70, Sun Microsystems Laboratories, Palo Alto, CA, December 1998.

[2] O. Agesen and D. Detlefs. Finding References in Java™ Stacks. In *Proceedings of the OOPSLA'97 Workshop on Garbage Collection and Memory Management*, Atlanta, GA, USA, October 1997.

[3] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. In *Proceedings of OOPSLA'99*, pages 207–222, Denver, Colorado, USA, November 1999.

[4] A. W. Appel. Garbage Collection can be Faster than Stack Allocation. *Information Processing Letters*, 25(4):275–279, January 1987.

[5] A. W. Appel. Simple Generational Garbage Collection and Stack Allocation. *Software — Practice and Experience*, 19(2):171–183, March 1988.

[6] J. F. Bartlett. Compacting Garbage Collection with Ambiguous Roots. Technical Report 88/2, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, February 1988.

[7] J. F. Bartlett. Mostly-Copying Garbage Collection Picks up Generations and C++. Technical Report TN-12, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, October 1989.

[8] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software — Practice and Experience*, pages 807–820, September 1988.

[9] C. J. Cheney. A Non-Recursive List Compacting Algorithm. *Communications of the ACM*, 11(13):677–678, November 1970.

[10] D. Detlefs and O. Agesen. Inlining of Virtual Methods. In *Proceedings of ECOOP'99*, pages 258–278, Lisbon, Portugal, June 1999.

[11] M. J. R. Goncalves and A. W. Appel. Cache Performance of Fast-Allocating Programs. Technical Report CS-TR-482-94, Princeton University, December 1994.

[12] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[13] M. S. Johnstone and P. R. Wilson. The Memory Fragmentation Problem: Solved? In *Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, Canada, October 1998. ACM Press.

[14] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Ltd, 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[15] H. B. M. Jonkers. A Fast Garbage Compaction Algorithm. *Information Processing Letters*, 9(1):26–30, July 1979.

[16] B. Lang and F. Dupont. Incremental Incrementally Compacting Garbage Collection. *ACM SIGPLAN Notices*, 22(7):253–263, July 1987.

[17] H. Lieberman and C. E. Hewitt. A Real-Time Garbage Collector based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, 1983.

[18] T. Printezis. Hot-Swapping between a Mark&Sweep and a Mark&Compact Garbage Collector in a Generational Environment. Technical Report TR-2001-78, Department of Computing Science, University of Glasgow, Scotland, March 2001.

[19] T. Printezis and D. Detlefs. A Generational Mostly-Concurrent Garbage Collector. In *Proceedings of the 2000 International Symposium on Memory Management*, pages 143–154, Minneapolis, MN, USA, October 2000. ACM Press.

[20] T. Printezis and D. Detlefs. A Generational Mostly-Concurrent Garbage Collector. Technical Report TR-2000-88, Sun Microsystems Laboratories, June 2000.

[21] P. Sansom. Combining Single-Space and Two-Space Compacting Garbage Collectors. In *Functional Programming, Glasgow 1991: Proceedings of the 1991 Workshop, Portree, UK*, pages 312–323. Springer-Verlag, 1992.

[22] Sun Microsystems Inc. Ultra™ 80 Workstation Profile. http://www.sun.com/desktop/products/Ultra80/ [November 5, 2000].

[23] Sun Microsystems Inc. *Java™ Object Serialization Specification — JDK™ 1.2*, November 1998. Revision 1.43.

[24] E. Ukkonen. On-Line Construction of Suffix-Trees. *Algorithmica*, 14(3):249–260, 1995.

[25] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[26] D. White and A. Garthwaite. The GC Interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories, 1999.

[27] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the First International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St Malo, France, September 1992. Springer-Verlag.

[28] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the Second International Workshop on Memory Management*, number 986 in Lecture Notes in Computer Science, Kinross, Scotland, September 1995. Springer-Verlag.

C Timing Results

GCBench	H-S	M&C	M&S
Elapsed Time (sec)	42.92	42.18	45.08
Total GC Time (sec)	31.74	30.94	33.96
Young Gen Size (MB)	2	2	2
Old Gen Size (MB)	32	32	32
Young GC Avg (ms)	32.63	30.27	34.11
Young GC Max (ms)	198.1	79.7	197
Young GC Total (sec)	24.70	22.92	25.82
Old GC Avg (ms)	292.57	333.11	324.94
Old GC Max (ms)	499.3	965.1	587.9
Old GC Total (sec)	7.02	7.99	8.12
M&Cs Performed	4	24	0
M&Ss Performed	20	0	25

Table 1: **GCBench** results.

MarkTest	H-S	M&C	M&S
Elapsed Time (sec)	15.10	17.60	14.58
Total GC Time (sec)	10.70	13.17	10.21
Young Gen Size (MB)	2	2	2
Old Gen Size (MB)	32	32	32
Young GC Avg (ms)	56.00	50.69	57.41
Young GC Max (ms)	103.81	69.54	101.45
Young GC Total (sec)	6.32	5.72	6.48
Old GC Avg (ms)	397.37	676.27	338.61
Old GC Max (ms)	645.54	702.90	345.18
Old GC Total (sec)	4.37	7.43	3.72
M&Cs Performed	2	11	0
M&Ss Performed	9	0	11

Table 2: **MarkTest** results.

GCOld	H-S	M&C	M&S
Elapsed Time (sec)	75.58	84.65	72.54
Total GC Time (sec)	38.78	47.84	35.77
Young Gen Size (MB)	2	2	2
Old Gen Size (MB)	350	350	350
Young GC Avg (ms)	15.09	14.75	16.22
Young GC Max (ms)	64.1	41.9	63.4
Young GC Total (sec)	19.5	19.05	20.95
Old GC Avg (ms)	6423	9588	4937.7
Old GC Max (ms)	9511.9	10727.7	5162.2
Old GC Total (sec)	19.26	28.76	14.81
M&Cs Performed	1	3	0
M&Ss Performed	2	0	3

Table 3: **GCOld** results.

JOS	H-S	M&C	M&S
Elapsed Time (sec)	147.17	149.03	147.70
Total GC Time (sec)	4.00	5.21	4.08
Young Gen Size (MB)	2	2	2
Old Gen Size (MB)	110	110	110
Young GC Avg (ms)	24.40	24.25	25.10
Young GC Max (ms)	59.9	59	63.8
Young GC Total (sec)	2.83	2.81	2.91
Old GC Avg (ms)	1169.8	2400.3	1167.4
Old GC Max (ms)	1169.8	2400.3	1167.4
Old GC Total (sec)	1.17	2.39	1.16
M&Cs Performed	0	1	0
M&Ss Performed	1	0	1

Table 4: **JOS** results.

DNA	H-S	M&C	M&S
Elapsed Time (sec)	203.37	208.32	209.45
Total GC Time (sec)	130.59	135.88	136.5
Young Gen Size (MB)	2	2	2
Old Gen Size (MB)	400	400	400
Young GC Avg (ms)	226.39	214.40	238.29
Young GC Max (ms)	333	308.4	342.4
Young GC Total (sec)	112.29	106.34	118.19
Old GC Avg (ms)	6097.5	9842.57	6099.77
Old GC Max (ms)	6448.2	9929.9	6442.8
Old GC Total (sec)	18.29	29.52	18.3
M&Cs Performed	0	3	0
M&Ss Performed	3	0	3

Table 5: **DNA** results.

Javac	H-S	M&C	M&S
Elapsed Time (sec)	224.63	223.77	224.20
Total GC Time (sec)	31.38	31.41	31.38
Young Gen Size (MB)	2	2	2
Old Gen Size (MB)	135	135	135
Young GC Avg (ms)	11.79	11.12	12.35
Young GC Max (ms)	38.3	38.1	38.8
Young GC Total (sec)	26.79	25.24	28.05
Old GC Avg (ms)	2282.4	3071.6	1649.25
Old GC Max (ms)	2824.8	3416.6	1726.5
Old GC Total (sec)	4.56	6.14	3.29
M&Cs Performed	1	2	0
M&Ss Performed	1	0	2

Table 6: **Javac** results.

Parallel Garbage Collection for Shared Memory Multiprocessors

Christine H. Flood
Sun Microsystems Laboratories
Christine.Flood@sun.com

David Detlefs
Sun Microsystems Laboratories
David.Detlefs@sun.com

Nir Shavit
Tel-Aviv University
shanir@tau.ac.il

Xiaolan Zhang
Harvard University
cxzhang@eecs.harvard.edu

Abstract

We present a multiprocessor "stop-the-world" garbage collection framework that provides multiple forms of load balancing. Our parallel collectors use this framework to balance the work of root scanning, using *static overpartitioning,* and also to balance the work of tracing the object graph, using a form of dynamic load balancing called *work stealing.* We describe two collectors written using this framework: pSemispaces, a parallel semispace collector, and pMarkcompact, a parallel markcompact collector.

1 Introduction

The Java™ programming language is increasingly used for large, memory-intensive, multithreaded applications run on shared-memory multiprocessors. Most Java virtual machines (JVM™'s) employ "stop-the-world" garbage collection (GC) algorithms that first halt the running threads and then perform the GC. If we have more than one processor available, it makes sense to employ them all in the GC process. This paper describes the parallelization of two sequential GC algorithms to allow them to take advantage of all available processors.

The Java Technology Research Group at Sun Microsystems Laboratories[1] has developed a JVM that includes a *GC interface* [12] to support multiple GC algorithms, thus enabling comparison of various GC strategies in a high performance virtual machine. This paper describes our augmentation of this interface with a parallel infrastructure to support multiple parallel GC strategies. We use this infrastructure to parallelize two well-known collection schemes: a two-space copying algorithm (*semispaces*) and a mark-sweep algorithm with sliding compaction (*markcompact*). The resulting algorithms have outperformed their highly-tuned product-quality sequential counterparts on multiprocessors.

In parallelizing sequential GC algorithms, one has to tackle two key issues: load balancing of all parts of the algorithm and re-engineering of any inherently sequential elements.

In both algorithms, the key to load balancing is to correctly and efficiently partition the task of tracing the object graph. This task unfortunately does not lend itself to static partitioning. Our approach, described in the following sections, is to combine static partitioning with dynamic load balancing based on work stealing. We show that this combination of static and dynamic methods leads to effective parallelization of both the semispaces and markcompact collectors. It is our belief that the effectiveness of our dynamic partitioning is the result of a finely-tuned lock-free work-stealing algorithm based on Arora *et al.* [1] whose low overhead allows us to balance our work at the individual object level.

In both algorithms there are parts that are not easily parallelizable. In the semispaces algorithm these included: installing forwarding pointers, allocating in parallel, and scanning the card table for references from the old generation. Though the installation of forwarding pointers is not parallelized, it is performed in a lock-free manner. We make allocation less of a sequential bottleneck by globally allocating local buffers from which objects may be allocated without synchronization. Scanning the card table requires a novel partitioning scheme to achieve good load balancing. In markcompact the

[1] www.sun.com/research/jtech

main inherently sequential part is the compaction phase, which involves copying all objects to one end of the heap. Here we statically partition the old generation heap into n partitions, compacting even partitions in one direction and odd partitions in the other direction, thus avoiding synchronization and optimizing the size of the free areas.

1.1 A short description of previous work

Endo *et al.* [11] describe a parallel stop-the-world GC algorithm using work stealing. Their algorithm depends on threads with work copying some work to auxiliary queues, where the work is available for stealing. Threads without work look for an auxiliary queue with work, lock the queue, and steal half of the queue's elements. Our work extends theirs by using a lower-overhead work-stealing mechanism, and by addressing the harder problem of parallelizing relocating collectors, not just a non-relocating mark-sweep algorithm.

Halstead [6] describes a multiprocessor GC for Multilisp. Each processor has its own local heap, and they use lock bits for moving and updating forwarding pointers. Load balancing is done statically rather than dynamically.

Many collectors operate concurrently with mutator activity [9, 4, 5, 8]. This kind of concurrency is orthogonal to the style of parallel collection we describe in this paper. A collector might combine both: some concurrent collectors have stop-world phases that might be performed in parallel, and collectors with concurrent GC threads might use several such threads working in parallel to cope with high aggregate garbage-creation rates in multi-threaded programs.

Steensgaard [10] explores a clever method for partially parallelizing collection. Compile-time analysis identifies allocation sites that allocate objects that never *escape* the allocating thread (are never accessible to other threads.) Such objects are allocated in a *thread-local heap,* which can be collected independently of other threads. This technique avoids the synchronization issues that general parallel collection must address, but requires extensive and expensive static analysis, and only a subset of objects may be collected thread-locally.

1.2 Overview

Section 2 presents basic parallel programming techniques. Section 3 presents our parallel GC infrastructure, which applies these techniques to the garbage collection problem. Sections 4 and 5 describe two parallel algorithms we implemented using this infrastructure: pSemispaces and pMarkcompact. Section 6 presents results for three benchmarks. Section 7 presents conclusions.

2 Parallel Programming Basics

If we had a predetermined amount of work to do and were able to partition it perfectly across all available processors, we would achieve perfect parallelism and finish the collection in the least possible amount of time. Some tasks can be partitioned in this way; we call them *statically partitionable.* Other tasks are difficult to divide into subtasks of predictable size. For example, tracing the graph of a program's live data is difficult to subdivide *a priori,* because it depends on the shape of the object graph. Many tasks fall somewhere in between: we are able to partition them statically into roughly, but not exactly, equivalent subtasks. We *overpartition* such tasks. That is, we break the tasks into more subtasks than we have threads, and then each thread dynamically claims one subtask at a time.

There are two motivations for overpartitioning. First, the number of processors available to the GC process is unpredictable due to load on the machine from other processes. If a task were divided into exactly n subtasks on an n-processor machine, and one of the processors were unavailable, then one processor would have to complete two subtasks, thereby doubling the time for the computation. With overpartitioning, this extra subtask would be divided into several smaller subtasks that may be distributed across the active processors. Second, when we only have a rough estimate of how much work each subtask represents, assigning just one task to each processor risks one of those tasks being significantly larger than the others. Overpartitioning both decreases this risk by making smaller subtasks, and enables processors that have finished smaller subtasks to take on additional work.

Some tasks are not even approximately statically partionable. These tasks require some form of dynamic load balancing. *Work stealing* [2] is a highly effective load balancing technique in such situations. In this approach, each thread works on its own tasks until it runs out of work, and then takes the initiative to steal work from one of the other processors.

2.1 A short explanation of lock-free work-stealing queues

Arora *et al.* present a non-blocking implementation of a double-ended queue data structure tailored

to support work stealing with minimal synchronization. Each thread has its own work queue of tasks. There are three fundamental operations: *PushBottom* pushes an element onto the bottom of the queue, *PopBottom* pops an element from the bottom of the queue, and *PopTop* pops an element from the top of the queue. *PushBottom* and *PopBottom* are local operations that usually require no synchronization. *PopTop* is used for stealing from other threads' queues.

A parallel algorithm using work stealing starts with available tasks distributed among the work queues. Each thread uses *PopBottom* to claim tasks from its local queue. Execution of this task may reveal new subtasks, which are then added to the local queue using *PushBottom*. When a thread runs out of work it uses *PopTop* to steal a task from some other thread's work queue. Synchronization is required only when stealing an element from another queue or when claiming the last element from the local queue.

We modified the algorithm of Arora *et al.* in several ways. We added a termination detection protocol to ensure that all work is complete before any thread terminates. We also added support for fixed size queues in the form of an overflow detection and handling mechanism.

3 Parallel GC Infrastructure

3.1 Balancing root scanning

Garbage collection computes the transitive closure of objects reachable from a set of *root* pointers. In our JVM, the root set consists of class statics, thread stacks, etc. We overpartition these roots into groups, and the GC threads compete dynamically to claim root groups. Even if the static partitioning succeeds in balancing root scanning, starting off with balanced groups is not sufficient. Some roots may lead to large data structures, while others may lead to single objects.

3.2 Balancing traversal of live data

We solve this problem by using work stealing to dynamically balance the load. The tasks are references to objects to be *scanned,* i.e., examined for pointers to other objects.[2] A scanning GC thread acquires an object reference either from its local queue or by stealing from another thread's queue, and pushes any outgoing references found in the object onto its local queue. The termination detection protocol is used to determine the completion of the transitive closure.

Consider the behaviour of this algorithm on a large linked data structure, say a binary tree. One thread will scan a root pointer referencing the top-level node of the tree, push both child nodes onto its work queue, and then pop one of the child nodes for processing. The other child node is now available for stealing. In this way, for a sufficiently large tree, the load will be dynamically balanced.

3.3 Termination detection

The termination protocol is based on a status word containing one bit for each thread participating in the GC. All threads start off marked active. As long as a thread has local work, gets work from the overflow lists (see section 3.4), or succeeds in stealing work, its bit in the status word remains on. Once it is unable to find work it sets its status bit to off and loops, checking to see if all the status bits are off. If so, then all threads have offerred to terminate, so the algorithm is complete. If not, the thread peeks at other threads' queues, attempting to find one with work to steal. If it finds a thread with work to steal, the thief sets its status bit to active and tries to steal the work. If it succeeds, it goes back to processing. If it fails, it sets its status bit back to inactive and resumes the loop.

Our colleague Peter Kessler has suggested replacing the status word with an integer indicating the number of active threads. To offer termination, a thread would decrement this count with an atomic instruction; if the count goes to zero, all threads have terminated. When an inactive thread becomes active, it would increment the count, again with an atomic instruction. This avoids the parallelism limitation imposed by the bit-width of a word, but we have not yet implemented this proposal.

3.4 Handling overflow in GC work-stealing queues

In order to avoid allocation during GC we allocate fixed-size work-stealing queues at startup time and use them for all GC's. This required modifications to the work-stealing code to check for overflow, and a mechanism for handling overflow gracefully by offloading some items to a global *overflow set.* Threads without work look to the overflow set for work before resorting to stealing. We wished to

[2] For large objects, especially large arrays of references, it might be advantageous to consider the object as comprised of several *chunks*, and subdivide the object-scanning task into the separate tasks of scanning each chunk. We have not implemented this extension.

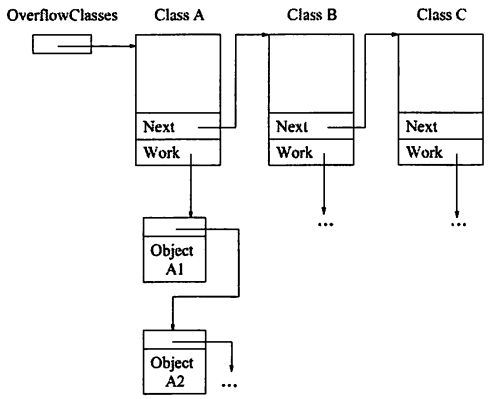


Figure 1: Overflow sets

be able to handle overflow without any additional storage space and also to avoid "thrashing" of objects between the overflow set and the work-stealing queues.

We modified *PushBottom* to check for possible overflow before adding an element. If adding an element would cause the queue to overflow, we pop all elements in the bottom half of the queue and add them to the overflow set.

The overflow set mechanism, due to our colleague Ole Agesen, exploits a class pointer header word present in all objects in our implementation. As shown in Figure 1, for each class X, we link all instances of X in the overflow set together in a linked list whose head is contained in the class structure for X. Each object's class pointer is overwritten with the "next" pointer of the list. This does not destroy information, since all objects in a given class's list are instances of that class. All classes with instances in the overflow set are linked into another list. This mechanism represents the overflow set with only a small per class storage overhead.

By draining only the bottom half of a queue on overflow and filling no more than the top half of the queue when retrieving work from the overflow set, we ensure that no object will be placed in the overflow set more than once, thus avoiding "thrashing."

4 Parallel Semispaces

Semispaces (a.k.a. "copying") collection divides the heap equally into two regions: from-space and to-space. Objects are allocated in from-space until it fills up, then a GC is triggered. Reachable objects are copied into a contiguous area of to-space, leaving the remaining space free for allocation. As the GC traces the transitive closure, it copies each object when it is first encountered, leaving a *forwarding pointer* in the from-space copy of the object to indicate its new address in to-space. Subsequent references to this object are updated with the forwarding pointer.

In the elegant style of Cheney [7], a *copy* pointer tracks the next free address, and a *scan* pointer tracks the next object to be scanned. The GC scans the object indicated by the scan pointer; it examines references in the object, copying any referenced object still in from-space to to-space, updating the copy pointer. The scan pointer is then updated to point to the next object. Collection is complete when the scan pointer reaches the copy pointer, at which point we swap to-space and from-space, and resume the program.

Our *pSemispaces* algorithm parallelizes this sequential algorithm. We depend on the infrastructure to properly distribute the process of scanning the roots. Rather than using Cheney's copy and scan pointers to represent the set of objects to be scanned, we use explicit work-stealing queues.

With a parallel copying collector, many threads allocate objects in to-space at the same time. One approach to managing this concurrency would be for each thread to increment the copy pointer atomically for each object it copies, using some hardware operation such as fetch-and-add or compare-and-swap (CAS) [3]. However, our experiments indicate that this results in too much contention. The alternative we adopted was to have each thread use such atomic allocation only to allocate relatively large regions called *local allocation buffers* (LABs). A thread can then do local allocations within this buffer with no synchronization. A thread can also deallocate its most recent allocation, which is useful in parallelizing the insertion of the forwarding pointer, as we explain below. LABs should be large enough to reduce contention on the copy pointer, yet small enough to avoid excessive fragmentation. Note that the potential fragmentation introduced by LABs makes it possible that to-space may not hold all of the objects copied from from-space. However, this is a concern only when the heap is very nearly full.

Collection must preserve the shape of the object graph. If several threads are simultaneously processing references to the same uncopied object in from-space, only one may succeed in copying the object. The others must observe that the object has been copied and update their references according to the forwarding pointer installed by the copying thread. We accomplish this by having each thread speculatively allocate space for the object in its LAB, and then use a CAS to update the from-space object's forwarding pointer to point to the speculative new

address. If the CAS succeeds, the thread proceeds to copy the object. If the CAS fails, the CAS returns the updated forwarding pointer.[3] The thread uses this value to update its reference, and then locally retracts its speculative allocation.

The semispaces algorithm is often used in youngest generations of generational collectors [7, 13]. A generational collector has two or more generations; objects are usually allocated in younger, smaller generations, and *promoted* to older generations if they survive long enough. The hope is that youngest-generation collections are significantly faster than collections of the entire heap, and likely to reclaim sufficient space to continue computation. However, multi-threaded programs running on multiprocessors will have larger aggregate allocation rates than single-threaded programs, and will therefore fill a young generation of a given size more quickly, increasing collection frequency. It is therefore attractive to increase the size of the youngest generation to reduce collection frequency with multi-threaded programs, and to use parallelism to keep pause times low and throughput high.

Two further issues must be addressed when using the pSemispaces algorithm in the youngest generation of a generational collector. First, collector threads will allocate both in to-space and in the older generation (for promotion). Both forms of allocation must be parallelized; old-generation promotion therefore uses the same LAB-based allocation technique as to-space allocation. Second, when performing a youngest-generation collection, we treat all older generation objects as roots. We cannot traverse the entire heap to find youngest-generation references, or else youngest-generation collection will be as costly as collection of the entire heap. Therefore, generational systems, including ours, often keep track of such old-to-young references using a *card table,* an array whose entries correspond to subdivisions of the heap called cards. When mutator code updates a reference field, it also "dirties" the corresponding card table entry. The youngest-generation collector scans the card table to find these dirty entries, which are the ones whose corresponding cards might contain old-to-young references.[4]

For large heaps, scanning the card table may take a long time and therefore should be partitioned across threads. At first we partitioned this work in the most straightforward way: dividing the card table into consecutive contiguous blocks, which were claimed by the GC threads. Unfortunately this didn't work well on some applications, because some blocks were very dense, while others were sparse; for example, large arrays of references caused dense blocks. Scanning the dense blocks was dominating the cost of the GC. To address this problem we instead overpartitioned the card table into N *strides,* each of which is a set of cards separated by intervals of N cards. Thus cards $\{0, N, 2N, ...\}$ comprise one stride, cards $\{1, N+1, 2N+1, ...\}$ comprise the next, and so on. This causes dense areas to be partitioned across tasks. As usual, threads compete to claim strides.

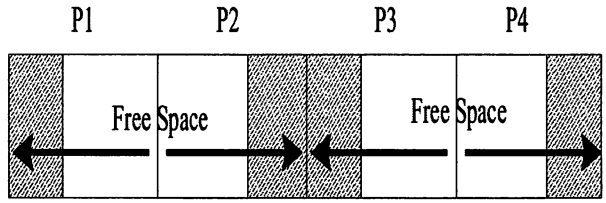


Figure 2: Parallel Compaction

5 Parallel markcompact

Our old generation uses a markcompact collector. The original sequential markcompact collector consists of four major phases:

- The marking phase, which identifies and marks live objects.
- The forwarding-pointer installation phase, which computes the new addresses live objects will have after compaction and stores these addresses as forwarding pointers in the objects' headers.
- The reference redirection phase, which updates references in live objects to the new addresses of the objects they reference.
- The compaction phase, which copies live objects to their new compacted addresses.

Our *pMarkcompact* algorithm parallelizes this single-threaded algorithm, by parallelizing each of the phases. The parallelization of the first three phases is relatively straightforward, but the final compaction phase presented difficulties. The original sequential compaction phase compacted all live data to the low end of the heap. In the parallel case, it was difficult to ensure that one thread did not overwrite object data that another thread had yet

[3] In CAS implementations of which we are aware.

[4] When a card contains old-to-young references after a collection, the collector leaves the corresponding card table entry dirty.

to copy. Our solution to this problem is to break the heap into n *regions,* where n is the number of GC threads. Each thread claims a region and slides live objects in its own region only. (Section 5.2 discusses the criteria that influence the selection of region boundaries.) The direction to which objects are moved alternates for odd and even numbered regions. Figure 2 shows an example of a heap with 4 regions and 2 free areas after compaction. In general, a heap with n regions has $\lfloor \frac{n+1}{2} \rfloor$ contiguous free areas. For practical purposes, a small number of sufficiently large contiguous free areas allows allocation as efficiently as a single free area.

The following subsections describes each parallel phase in detail.

5.1 Parallel marking

Similar to pSemispaces, the parallel marking phase employs the parallel GC infrastructure to statically partition the root set and to dynamically balance further marking work through work stealing. Each thread keeps a work queue of objects to be scanned for pointers to other objects. When a thread runs out of objects, it attempts to steal an object from the work queue of another thread. Unlike pSemispaces, which requires synchronization on the installation of forwarding pointers, marking is idempotent and therefore requires no synchronization.[5]

5.2 Parallel forwarding-pointer installation

At this point, all live objects have been marked. The next phase corresponds to the "sweep" phase of a mark-sweep collector, and also has the side-effect of computing the distribution of live data, which will guide the partitioning of the heap into the regions discussed above. First, we overpartition the heap into m *units* of (roughly) equal size. (We ensure that unit boundaries are object-aligned, which leads to the approximation above.) The value of m is typically $4n$, where n is the number of GC threads. The GC threads compete to claim units; for each unit, the thread traverses the objects, counting the number of bytes of live data in the unit, and coalescing contiguous regions of dead objects into single blocks traversable in constant time.

When all units are processed, we know the exact amount of live data in each unit, and can partition the heap into *regions* with approximately equal amounts of live data. The partition is such that each region contains one or more of the units created in the previous pass, i.e. regions are unit-aligned. Regions are the partitions used to solve the compaction problem; they are the heap divisions in Figure 2. The region that contains an object dictates the direction in which it will be copied. Since we know how much live data is in each unit in a region, it is straightforward to calculate the new address of the first live object in a particular unit, by summing the live data of the previous units in the region (in the appropriate compaction order for the region). Thus, forwarding pointer installation can use the unit partitioning already established. GC threads dynamically claim units and install forwarding pointers in all live objects within the unit.

[5]Note that this lack of synchronization also depends on having the mark bits present in the object; if an external marking array were used then one word might contain several marks, which would necessitate synchronization.

5.3 Parallel reference redirection

Redirecting object references requires scanning roots, objects in the current generation, and objects in other generations for references to objects in the current generation. The forwarding pointers inserted by the previous phase are used to update these references. We rely on the parallel GC infrastructure to balance the work of scanning the roots. Currently, the scanning of the young generation is treated as a single task; in the future, this might be further partitioned. Within the old generation we reuse the previous unit partitioning.

5.4 Parallel compaction

The last phase is parallel compaction. As discussed previously, we use the larger-grained region partitioning in this phase. There is a trade-off here between parallelism, which favors more, smaller partitions, and allocation efficiency, which favors fewer, larger partitions (and thus, fewer, larger free areas at the end of compaction.) We currently favor allocation efficiency, by making the region partition an exact partition (as opposed to an overpartition.) This design choice will be investigated further in the future.

6 Results

6.1 Benchmarks

We present results for three benchmarks. **GCOld** is a synthetic program which can be used to present a variety of loads to a garbage collector, including large heaps requiring significant old-generation

collections. **SpecJBB** is a scalability benchmark inspired by TPC-C which emulates a 3-tier system with emphasis on the middle tier. **Javac** is a compiler that translates Java programming language source code to Java class files.

The **GCOld** application allocates an array, each element of which points to the root of a binary tree about a megabyte in size. An initial phase allocates these data structures; then the program does some number of *steps*, maintaining a steady-state heap size. Each step allocates some number of bytes of short-lived data that will die in a young-generation collection, and some number of bytes of nodes in a long-lived tree structure that replaces some previously existing tree, making it garbage. Each step further simulates some amount of mutator computation by several iterations of an busy-work loop. Finally, since pointer-mutation rate can be an important factor in the performance of generational collection, each step modifies some number of pointers (in a manner that preserves the amount of reachable data). Command-line parameters control the amount of live data in the steady state, the number of steps in the run, the number of bytes of short-lived and long-lived data allocated in each step, the amount of simulated work per step, and the number of pointers modified in a step. We ran **GCOld** with 300MB of live data, allocating three bytes of short-lived data for every byte of long-lived data.

SpecJBB is a "throughput-based" benchmark: it measures the amount of work accomplished in a fixed amount of time, rather than the amount of time required to accomplish a fixed amount of work. To create runs that can be compared to determine parallel speedup for GC, we run with a fixed number (8) of "warehouses" (i.e., mutator threads), and considered only the first 500 collections of each run. We believe the mutator behavior between these collections is sufficiently similar to make these runs comparable.

Each graph is annotated with heap configuration parameters used for the runs. A heap configuration specifies the sizes of the young and old generations (which are fixed in all our experiments.) For example, `16m:600m` indicates a young generation of 16 MB and an old generation size of 600 MB. The number of young- and old-generation collections is similar across all runs, including the sequential run, since allocation behavior is largely unaffected by collection algorithm. (We discuss an exception below.)

The runs were performed on a Sun Enterprise™ 3500 server, with 8 336 MHz UltraSPARC™ processors sharing 2 Gbyte of memory. The collector we ran was a generational collector with a parallel semispaces young generation and a parallel markcompact old generation.

6.2 Scalability

Figure 3 presents our results in terms of scalability graphs. The x-axis is the number of processors. The y-axis shows speedup relative to the performance of the parallel collector run on one processor. We also show the curve for linear speedup and the performance of the sequential form of each GC algorithm. Speedups for the young generation and old generation are shown on separate graphs; speedups are calculated on the basis of total time for collections of the given type.

Table 1 gives the average and total GC times for the sequential runs and the parallel runs with one and eight processors.

	seq	par(1)	par(8)
GCOld			
young avg (ms)	216	298	53
young total (s)	211.35	290.83	53.27
old avg (ms)	13920	19498	3632
old total (s)	876.98	1228.42	221.58
SpecJBB			
young avg (ms)	204	255	54
young total (s)	99.59	124.29	26.69
old avg (ms)	1880	3315	761
old total (s)	26.32	46.41	10.66
Javac			
young avg (ms)	33	40	10
young total (s)	2.26	2.69	.66
old avg (ms)	421	524	160
old total (s)	1.68	2.09	.96

Table 1: Average and total collection times

6.3 Discussion

We outperform the sequential algorithm using only two processors in most cases. The only case where we required 3 processors was in pMarkcompact for **SpecJBB**. Our hypothesis is that this is due to an optimization present in the sequential markcompact collector which we have not yet adapted to the parallel version. This *dense prefix* optimization avoids copying large blocks of data when there is only a small amount of free area to be reclaimed. In many applications this optimization eliminates a significant fraction of markcompact copying costs. We hope to adapt this technique to realize similar savings in the parallel version.

We achieve speedup factors on 8 processors of between 4 and 5.5, with the exception of the old-generation collections of **Javac.** One reason for this is that that there were 6 old-generation collections in the 8-processor run, but only 4 in the 1-processor and sequential runs. We believe that this increase is caused by fragmentation introduced by parallel LAB allocation during young-generation collection, and thus is an inherent cost of parallel collection. Note, however, that **Javac** has by far the smallest heaps of the benchmark runs. In larger problem sizes this effect is much less significant.

In the parallel mark-compact collector, we can measure the scalability of the individual phases separately. It turns out that all phases scale about as well as overall collection. For example, in **SpecJBB**, the overall 8-processor old-generation speedup is 4.351, and the speedups of the individual phases range from 3.7, for installing forwarding pointers and redirecting references, to 5.0 for sweeping. So no particular phase stands out as a clear scalability bottleneck. Still, clearly further work is needed to attempt to increase scalability (or explain the factors that inhibit it).

7 Conclusions

After exploring parallel techniques, and implementing two parallel collectors, we believe that there is great potential for improving both pause times and throughput using parallelism.

Large multi-threaded applications are being written in garbage-collected languages. These applications require heaps in the gigabyte range and beyond. Sequential GC algorithms will become an ever-greater scaling bottlneck. If systems intended to support such applications stop all threads for garbage collection, they must use parallel techniques to avoid this bottleneck.

8 Trademarks

References

[1] Nimar S. Arora; Robert D. Blumofe; and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[3] The SPARC Architecture Manual Version 9, Sun Microsystems, Inc.

[4] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M.Steffens. On-the-fly garbage collection: An exercise in cooperation. *CACM*, 21(11):966–975, November 1988.

[5] Damien Doligez and Georges Gonthier. Portable unobtrusive garbage collection for multiprocessor systems. In *Proceedings of the 1994 ACM Conference on Principles of Programming Languages*, pages 70–80, 1994.

[6] Robert H. Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In *1984 ACM Symposium on LISP and Functional Programming*, pages 9–17, New York, NY, 1984. ACM.

[7] Richard Jones and Rafael Lins. *Garbage Collection Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1996.

[8] John R. Ellis; Kai Li; and Andrew W. Appel. Real-time Concurrent Collection on Stock Multiprocessors. Technical Report 25, Digital Equipment Corporation Systems Research Center, February 1988.

[9] G. L. Jr. Steele. Multiprocessing compactifying garbage collection. *CACM*, 18(9):495–508, September 1975.

[10] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. *ACM SIGPLAN Notices*, 36(1):18–24, January 2001.

[11] Toshio Endo; Kenjiro Taura; and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. *Proceedings of High Performance Networking and Computing (SC97)*, 1997.

[12] Derek White and Alex Garthwaite. The GC interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories, 1998.

[13] Paul R. Wilson. Uniprocessor garbage collection techniques. *International Workshop on Memory Management, Springer-Verlag Lecture Notes in Computer Science*, 1992.

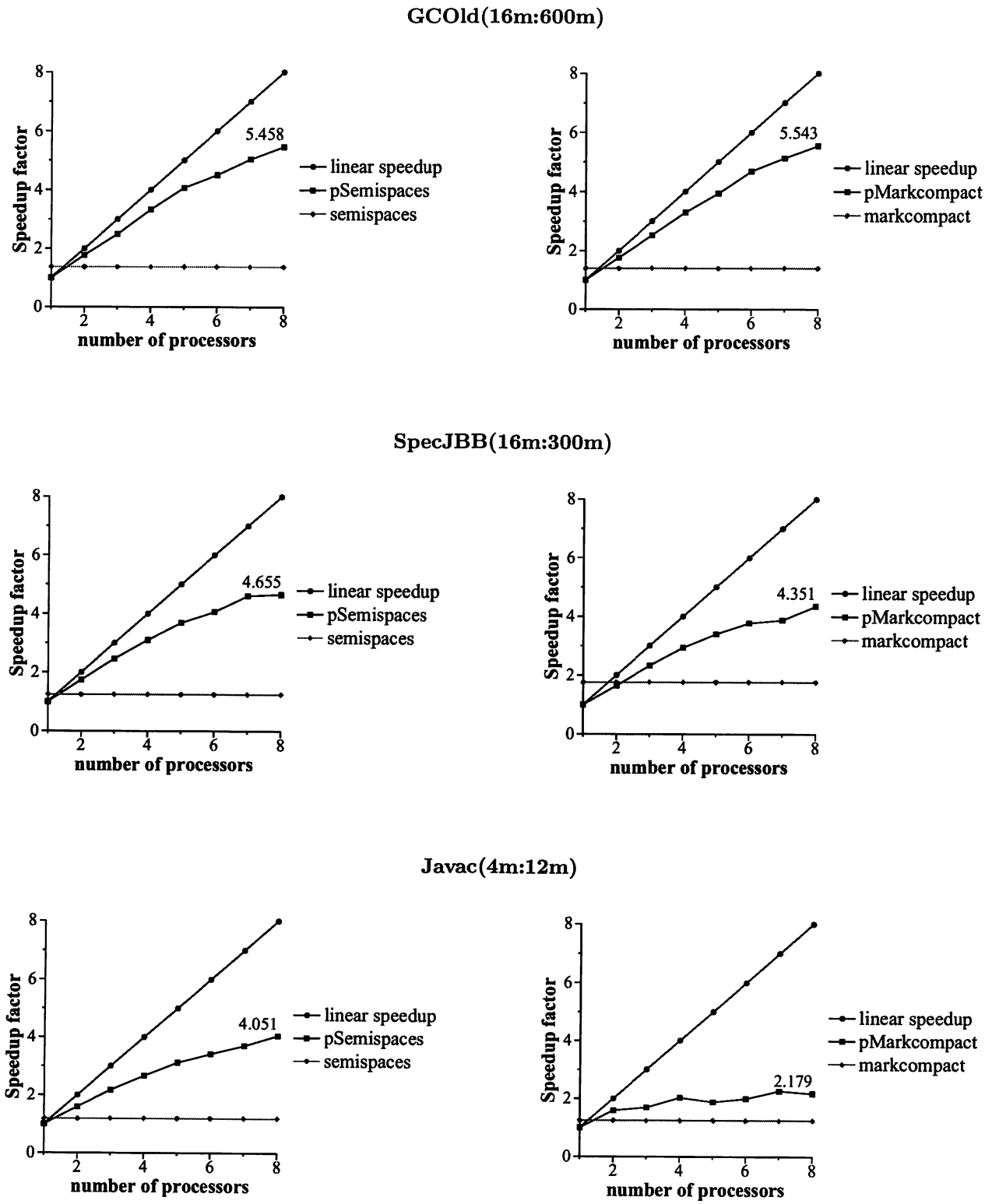


Figure 3: Speedup graphs

Automatic Persistent Memory Management for the Spotless Java™ Virtual Machine on the Palm Connected Organizer*

Daniel Schneider†, Bernd Mathiske, Matthias Ernst†, Matthew L. Seidl
Sun Microsystems, Inc.
901 San Antonio Road
MS MTV29-112
Palo Alto, CA 94303-4900
Daniel.Schneider@coremedia.com, Bernd.Mathiske@sun.com,
Matthias.Ernst@coremedia.com, Matthew.Seidl@sun.com

Abstract

Palm organizers are widely used in a multi-tasking fashion. *Users* switch from one application to another without losing the context established in either of them. Despite its obvious usefulness there is no *automatic* support for this convenience in the organizer's operating system, PalmOS. *Programmers* must implement event callbacks that use the PalmOS database API to save and reload specific application state. In this report we describe how this burden can be eliminated.

We enhanced the Spotless Java™ virtual machine for the Palm organizer with transparent multi-tasking support that automates persistence.

As a consequence, running Java programs can be transfered between two Palm organizers using the infra-red link. A transfered program will resume on the receiving organizer in the exact same state as it had on the sender. In addition, a HotSync operation can effectively be used to establish a checkpoint for each Java program involved.

Originally, the address range available for running programs was restricted to a few tens of KB in the *dynamic* RAM area. By directly addressing the much larger *static* RAM area our modified VM supports address ranges of several MB.

We provide an easy-to-use protocol that leverages persistent threads for automatic life cycle control of *external* resources (e.g. windows, forms and databases). When applied at the library level, this protocol maintains complete persistence transparency for the application programmer.

1 Introduction

The Spotless VM is a Java™ Virtual Machine for the Palm connected organizer that has been developed at Sun Microsystems Laboratories with the goals of small size, portability and readability of its source code [Taivalsaari et al., 1999]. A version of Spotless was then further developed to become the KVM [Sun, 1999], an extremely lean implementation of the Java Virtual Machine for use in devices that have very restricted physical memories (tens of kilobytes). The KVM is one of the core elements of the Java™ 2 Micro Edition (J2ME), which is targeted at consumer electronics and embedded devices. It has been ported to and is supported on a variety of platforms.

The Spotless VM runs only on PalmOS, the operating system of Palm connected organizers. It is a research vehicle at Sun Microsystems Laboratories for exploring new ideas and technologies for program-

†now at: CoreMedia AG, Düsternstrasse 3, D-20355 Hamburg

ming on small devices. In this report we describe how we enhanced the Spotless VM with memory management capabilities that automate persistence, that is, saving and restoring of code, objects and threads.

The Palm users' experience is that an application's state is saved when another application is invoked and that said state is restored when the former application is resumed. Given that the device's RAM is battery-backed, this behavior seems only natural to users. However, this advantage over desktop systems is only apparent to users – not to developers.

Application switches can happen at any point during program execution. Hence, a Palm application must be able to handle persistence at any point. The operating system, PalmOS, offers support by invoking specific event callbacks before every application suspension and resumption. Application programmers must implement these callbacks to provide the illusion of multi-tasking.

When writing suspension and resumption code, application programmers have to deal with the traditional dichotomy between primary and secondary memory. Only 64 to 256 K bytes of the Palm's RAM are freely accessible and available to represent dynamic application program state. The OS, using hardware memory protection, guards *all* write accesses to the larger part of the RAM and imposes a simple database API.

Unfortunately, a single write access through the database API can take 100ms or longer – the more items stored in the database, the slower the access. This leaves programmers with no choice but to carefully write significant amounts of code for the purpose of storing and recovering long-lived data – an error-prone task that repeats as new applications are developed and existing ones are modified and extended. This process very much resembles the work done by a multitasking OS or a thread library when performing a context switch, namely to save volatile processor state to memory.

In order to gain efficient write access, we bypass the database API by calling a undocumented PalmOS function that disables the memory protection. Thereupon all RAM on the device has the same access performance. By representing application stacks and heaps directly in the newly unprotected RAM area, we manage to eliminate the distinction between primary and secondary memory and to make all program state persistent.

In the next section, we describe relevant features of the Palm device and operating system. In section 3 we then present our implementation of persistence automation. We cover memory management, program life cycles, and a protocol to integrate external state that is not under direct control of the runtime system. Section 4 discusses the safety issues that are introduced with our solution. Section 5 presents the previous work most closely related to this research, and Section 6 prevents our conclusions.

2 The Palm Connected Organizer

Due to their relatively restricted requirements, handheld devices generally have quite different architectures than desktop systems. In this section, we give a brief overview of the Palm platform and the PalmOS operating system for readers who are not familiar with either.

2.1 The Device

The Palm Organizer is equipped with a Motorola 68000 compatible processor (the 16 MHz Dragonball MC68EZ328 in the case of the Palm V) and from 2 to 8 MB of RAM backed by the battery such that its state is preserved even if the device is switched off. The display has 160 by 160 pixels.

2.2 The Operating System

PalmOS is a simple, single-tasking operating system with a GUI widget set that enables the application developer to build event driven, graphical applications.

The 2 to 8 MB of physical RAM is divided into two parts:

- 64 KBytes of "dynamic memory" in older version of PalmOS, and up to 256 KBytes in more recent versions. This is the amount of RAM in which a PalmOS application executes. Memory in this area can be acquired from the operating

system through the `MemPtrNew` system call and then be freely written and read.

- The remaining memory – called "static memory" – is battery-backed. This area is maintained by the OS as secondary memory. It is organized as a list of "databases" with mostly untyped records of variable size to which the user can obtain a handle using the `DmGetRecord` call. By locking such a handle using the `MemHandleLock` call, a pointer to a record's location can be obtained. As long as a handle is locked the operating system guarantees not to move the associated record in memory. The pointer returned by `MemHandleLock` only allows read access to the record's memory area. Writing has to be performed using the `DmWrite` call – direct write access will result in a fatal exception and will eventually reset the device.

This partitioning of the device's physical memory can be avoided by using the undocumented system calls `MemSemaphoreReserve/MemSemaphoreRelease`. These will switch the memory protection of the "static" memory area off and on. When these calls are used, great care has to be taken when issuing other system calls as some may need the semaphore. This forces the programmer to release the memory semaphore before entering certain system calls.

As mentioned in the introduction, PalmOS has no automatic multi-tasking capabilities. If a user changes from one application to another, the first one will be asked to stop by an `AppStop` event. Once it has stopped, all dynamic memory will be reclaimed by the operating system. The second application will then be started using the same dynamic memory as the first. The operating system does not provide any implicit support for saving the context of an executing application. Instead, the application programmer is forced to save the state explicitly, in a database residing in static memory.

3 Automation of Persistence

Most computing environments distinguish transient primary memory (RAM) and stable secondary memory, typically on hard disks. Because the latter normally has much slower access characteristics, programs operate in primary memory and persistence is achieved by copying to secondary memory and restoring from there.

PalmOS treats dynamic RAM as primary memory and static RAM as secondary memory. But on the Palm device the *physical* access speed of dynamic and static RAM are exactly the same! Only because PalmOS controls accesses to static RAM by certain exclusive access functions does it appear to behave like "typical" secondary memory.

By circumventing this control, though, and making more direct use of static RAM, we can eliminate the distinction between primary and secondary memory. Hence, no data has to be copied between the two areas when a program is suspended.

The Spotless VM was first modified to have a "persistent heap," which is directly allocated in stable memory. The bytecode interpreter still uses dynamic memory, but all of the Java data (including bytecodes and threads) is allocated on the Java heap and thus in static memory. In order to be able to freely write to the heap, the VM uses the system call `MemSemaphoreReserve`, holding the semaphore during the normal interpretation of bytecodes and only releasing it when entering certain system calls that require it.

However, allocating the heap in "static" memory does not unburden the VM from having to shut down and resume as a Palm program, as discussed in section 3.2. Also, this "in place execution" inhibits the possibility of making a snapshot of a heap and then continuing execution without overwriting the snapshot data. We still chose this implementation because it greatly speeds up the shut down process as almost no data has to be copied to stable storage. Snapshots can still be achieved by stopping an application and then transferring the persistent store to a desktop system using the Palm device's serial port.

3.1 Store Structure

Although we eliminate the distinction between primary and secondary memory for objects, we borrow the notion of a "store" from persistent object caching systems to describe the entire durable memory image of an application. A Spotless store captures the execution state of a running program, its

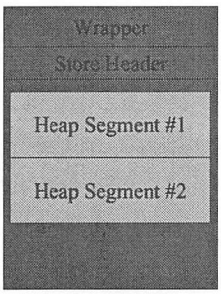


Figure 1: Store Structure

heap, and additional boot information necessary to resume the virtual machine at the point where it was interrupted.

Such a persistent store is represented by a PalmOS resource database of the type `'appl'`. This type tag causes the application manager to display the store like any other application installed on the Palm. Figure 1 illustrates the basic structural components of a store:

1. The wrapper, which contains a minimal boot program (less than 1 KB in size) that simply locates the Spotless VM program on the device and then starts it, passing the user-indicated store as a parameter.

2. The store header, which contains all global C variables that cannot be re-established from the context. These are mostly the various roots of the system, that is, references to objects located in the segmented heap. It also contains pointers to the first and the last of the resource database records that represent the store contents. The store header itself is stored in a resource record of type `'STOR'`.

   Figure 2 shows the definition of the associated C structure. The field `externalManager` will be explained in more detail in 3.4.

3. The segmented persistent heap. This is the area where objects are placed by the allocator and moved or deleted by the garbage collector. Each segment is stored in a resource database record of type `'VMem'` and is 64KB in size (the largest contiguous area that can be allocated in any Palm database).

   Each record has its own header block containing status information for the segment. Figure 3 shows the definition of the C structure that contains the header data.

```
struct storeStruct {
  int       lastRecordId;           // ID of the last record
  RECORD    first;                  // First record of this store
  RECORD    last;                   // Last record of this store
  RECORD    nextAllocation;         // Record in which the next
                                    // object allocation should take place
  cell *    bitmap;                 // Image of the Palm screen
                                    // when the store was suspended
  INSTANCE  externalManager;        // Singleton manager of external data

  // These are VM roots and globals:
  THREAD    UP;                     // List of all threads in the VM
  int       nActiveThreads;         // Number of active threads
  BYTE *    tagStack;               // Marks pointers on the current stack
  ...
  int       elapsedTime;            // The time the store was up
};
```

Figure 2: C Data Structure Representing the Store Header

```
struct recordStruct {
  int       id;               // The database ID of this record
  RECORD    next;             // The next record in the list or NULL
  int       size;             // The number of cells in this record
  cell *    bottom;           // Bottom of this record
  cell *    top;              // Top of this record
  CHUNK     firstFreeChunk;   // Allocate the next object at this location
  int       bumped;           // TRUE when the underlying MemBlock
                              // starts 2 bytes before this record
  cell *    breakTable;       // Break Table used
  cell      breakTableSize;   // by the garbage collector
  int       address;          // The address where this record was located
                              // when the store was suspended
};
```

Figure 3: C Data Structure Representing a Heap Record

The segmented heap is represented by a linked list of records using the `next` pointer to locate the following one.

The VM requires all heap objects to be located at a 32-bit aligned address, whereas the operating system may return addresses that are only 16-bit aligned. This problem is solved by actually requesting 16 bits more than needed to hold the record, which leads to one of the following situations.

If the memory area returned by the OS is not 32-bit aligned, the record is "bumped" up to the next 32-bit aligned address. When an application is suspended, the fact that the record was bumped is indicated by setting its header's `bumped` field to `TRUE` and the two bytes before the record to `0xFFFF` (Figure 4). This will be recognized by the VM when it resumes the application, and then it can properly align the contents of the record as described further in section 3.2.

If the memory area returned by the OS is 32-bit aligned, the record is simply placed at the beginning of that memory area leaving 16 bits at the end of it unused.

3.2 Lifecycle of a Persistent Program

Program execution comprises the following stages:

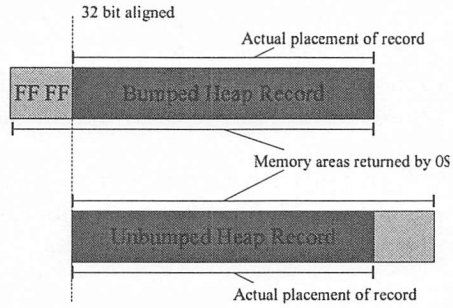


Figure 4: Bumping of Unaligned Addresses

Creation: A program is first started and a new store is created.

Execution: The program runs.

Suspension: The user switches to another application. The current application is interrupted and made persistent.

Resumption: The user resumes a previously suspended application. Execution will continue from the exact same state where it was left off.

Destruction: The computation terminates. This can be caused either by calling `System.exit`, terminating the last non-daemon thread or by not catching an exception. The store is then deleted.

The next sections will describe these four stages and associated actions taken by the VM in more detail.

3.2.1 Creation

When a new computation is started the user must supply a name for the store that will be created. The VM will then create a new PalmOS resource database of type `'appl'` and will first copy the contents of the wrapper database (`Wrapper.prc`) to the beginning of the newly created store.

Then the VM creates a store header record and a first heap record and initializes all fields. These records remain locked allowing the VM to access them directly.

3.2.2 Execution

The limited size of Palm database records dictates a two-level allocation scheme. When in the course of program execution the amount of available memory in the current heap segments becomes insufficient despite garbage collection efforts, another record is allocated and locked in memory. New records are appended to the end of the linked list of heap records.

Failure of record allocation is signaled by an exception.

3.2.3 Suspension

When the user changes to another Palm application, the VM automatically saves the current state of the computation by closing the persistent store in a controlled manner:

1. All objects whose state is partly transient (i.e., not located on the persistent heap) are requested to internalize their current external state. This process is further explained in 3.4.
2. Individual modules of the VM, such as the event handling system, are requested to memorize their state. This is implemented by copying C variables either to the heap or to appropriate fields in the store header.
3. All heap records and the store header are unlocked and can then be freely moved by the operating system.
4. The store database is closed. It will now appear as a normal PalmOS application in the application manager.

3.2.4 Resumption

In order to continue a computation from a persistent store, the user simply selects it in the application manager. The operating system will then invoke the wrapper code at the beginning of the store which will in turn locate the SpotlessVM and invoke it, passing the store as a parameter. The VM will then perform the following steps in order to resume the suspended computation.

First, it will open the store database and map the store header to the appropriate C `struct` (see Section 1). The store header is the only record of type `'STOR'` in the database and can thus be uniquely located by the VM.

Next, the VM will start opening the individual heap records – these can be identified by their record ID (ranging from 1 to `lastRecordId`) and type (`'VMem'`). After opening and locking a record, the header information has to be updated in case the record has been moved since it was suspended. This operation involves three major changes to the record header.

1. **Adjust the "bump" state.** If the record has been moved from an aligned to an unaligned address or visa versa, its bump state has changed. The new bump state can be obtained from the record's new memory address. The old bump state can be detected by reading the first two bytes of the record's new memory area. If these contain `0xFFFF` the record was bumped when the store was suspended. If the old and the new state differ, the record has to be moved by 16 bits to a 32-bit aligned memory address. Note that the required space is always available as the initial allocation requested 16 bits more memory than necessary to hold the heap record (Figure 4).

2. **Adjust header fields.** If the record has been moved, all pointers from the header into the heap segment need to be adjusted. The old location of the record can be read from the header field `address` and be compared to the record's new location.

3. **Append an entry to the break table.** We build on an improved version of the SpotlessVM with a compacting garbage collector that uses a break table [Haddon and Waite, 1967] to update pointers to relocated objects. We reuse the garbage collector code to adjust pointers to objects located in heap records that have moved (see 3.3 for more detailed information on the garbage collection algorithm). At this point the VM simply has to append an entry to the global breaktable to reflect the movement of the current record.

After all records have been opened and the appropriate break table entries have been appended, the VM runs a (slightly modified) pointer update phase of the garbage collector in order to adjust the pointers within the heap and from global C variables (still being stored in the store header) into the heap.

After all pointers have been updated, individual modules of the VM are asked to restore their state from the contents of their associated store header field.

Then callbacks that have been registered to recreate external state are executed (see 3.4).

Finally, the virtual machine resumes the execution of the application from the point where it was suspended earlier.

3.2.5 Destruction

When a program terminates – either by calling `System.exit`, terminating the last thread or by not catching an exception, the VM simply deletes its entire store. In case of abnormal termination this may seem a bit harsh, but considering that the store would be useless as it cannot be restarted and – without a tool like a store browser – cannot be analyzed for errors, it seems the right choice. As mentioned earlier, checkpoints of a store can and should be achieved by hotsyncing a store to a desktop system. This behavior could be changed when tools exist to do a post-mortem on programs that exit abnormally.

3.3 Memory Management

Store records can be moved in memory by the operating system, when the store is inactive. To compensate for this, as mentioned in the previous section, pointers need to be updated. This requires the VM to be able to tell the difference between a pointer and scalar data at any point where a pointer adjustment may take place. Otherwise, say, an `int` could be misinterpreted as a pointer and would thus be modified after an object (or a record) was moved in memory – leading at best to an erroneous computation.

Being able to tell *exactly* which values are pointers and which are scalars is referred to as *exactness* or *type accuracy*. It can be achieved by:

- keeping separate reference maps indicating pointer locations,
- directly tagging each individual value with at least one bit of type information.

The Spotless VM's memory management identifies a heap cell's type by analyzing the class of the object that the cell belongs to. Thus an object's class stores which fields contain pointer data and which scaler data. To identify a stack cell's type, the VM keeps a parallel stack containing the required type information. The secondary stack is called a tag stack, as it keeps a tag corresponding to each stack value and whither its a pointer or not. Each time the execution stack is modified, the tag stack is updated accordingly.

The garbage collector consists of a mark-sweep algorithm with sliding compaction that uses a break table to update pointers to moved objects [Jones and Lins, 1996]. Although this algorithm is relatively slow, it has two crucial advantages:

1. It does not need any extra space during the compaction/update phase. The break table is built in space that moved objects leave. This is important, as on a very memory limited device, there may not be enough additional memory available for many other garbage collection algorithms.
2. It can adjust pointers "into" any part of objects, not only pointers to the beginning of objects. We inherited this requirement from the original design of the Spotless VM, which features some pointers "into" objects for performance reasons (e.g., a method is directly referenced from a stack frame when executed, but it is not represented by its own heap object; instead, it is part of a method table heap object).

The current garbage collection algorithm does not move objects between heap segments. Although it is a compacting collector, this means there is still some fragmentation. Furthermore, when a computation is suspended, the VM will almost never find an empty segment that could be returned to the operating system. We have considered alterations to the algorithm to make it "multi-segmented", but would suggest that later implementations should instead take advantage of the segmentation and implement a generational collector.

3.4 External Data

When a program communicates with its environment, this often involves the creation of some external state that is not under the control of the program's runtime system. In case of the Spotless VM this typically concerns PalmOS structures such as windows, forms or databases. In this section, we introduce a protocol that provides automatic control over the life cycle of external state.

In order for external state to use our protocol, it has to be encapsulated in an instance of a developer-defined class that implements the interface `External` as shown in figure 5. We then regard an `External` as consisting of two sub-states:

External state: data external to the persistent store and references thereof. This also includes attributes representing operating system structures, such as a file descriptor or a form handle that will usually be invalid after the store has been suspended.

Internal state: a representation of the external state that is not dependent on any data structures outside the control of the persistent store. The `External` must be able to reconstruct its exact external state from the internal state including, for example, the read/write position of a file handle.

An `External` has to synchronize its internal state with its external state when the store is suspended and vice versa when it is resumed. In order for this to be performed in a thread-safe manner Spotless was adapted to use a protocol adopted from the Tycoon-2 system [Weikard, 1998, Gawecki and Wienberg, 1998] and implemented in the `ExternalManager` singleton class.[1]

All state transitions of `Externals` are controlled by a global `ExternalManager`. By adhering to this control center's protocol, the external state of a computation can be internalized on suspension and reestablished on resumption in a manner that prevents inconsistencies. Figures 5 and 6 show the pertinent interfaces.

[1] Tycoon-2's protocol for external resources builds on the basic protocol of Tycoon-1, which was first sketched in [Mathiske et al., 1995, Mathiske et al., 1997] and described in detail in [Kornacker, 1995]. The latter also influenced the API and protocol presented in [Jordan and Atkinson, 1999].

```
public interface External {
  public void createExternal() throws ExternalException;
  public void internalizeExternal() throws ExternalException;
  public void recreateExternal() throws ExternalException;
  public void destroyExternal() throws ExternalException;
}
```

Figure 5: The Interface for `External` Resources

```
public class ExternalManager {
  // "Freezes" the execution state of the VM:
  private native void stabilizeStore();
  public synchronized void stabilize();
  public synchronized void createAndRegisterExternal(External x)
      throws ExternalException;
  public synchronized void unregisterAndDestroyExternal(External x)
      throws ExternalException;
}
```

Figure 6: Major Methods of the `ExternalManager` Class

All four methods shown in Figure 5 must be implemented by an `External` to participate in the `External` protocol. By calling the method `createExternal`, the `ExternalManager` asks the `External` to create its external state. An `External` representing a database would, for instance, issue the appropriate OS call to open the database. The call `internalizeExternal` tells the `External` to internalize all external state, e.g., read the position in a database into an slot so the current state can be reestablished later. By calling `recreateExternal`, the `External` is asked to recreate its external state from its current internal; and by calling `destroyState`, the `External` is asked to destroy its external state, e.g., close the associated database.

The above four methods are invoked exclusively by the `ExternalManager` during different state changes of the persistent store. This means that in order to create external state, the `External`

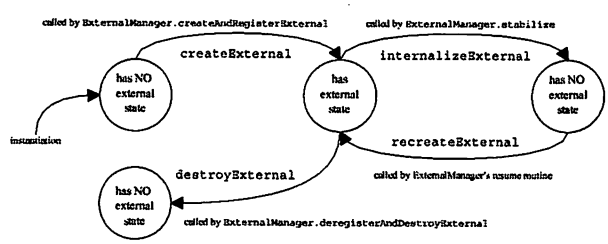


Figure 7: Life cycle of an `External`

has to register with the `ExternalManager` using the call `createAndRegisterExternal`. The `ExternalManager` will in turn call back to the `External` method `createExternal`. Unregistration is regulated in a similar manner.

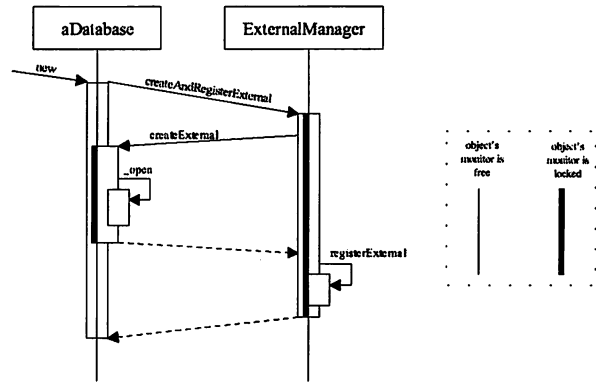


Figure 8: Registering with the `ExternalManager`

Figure 7 shows the major state transitions from the perspective of an `External`. Figures 8 and 9 show examples of a database class registering and unregistering with the `ExternalManager`.

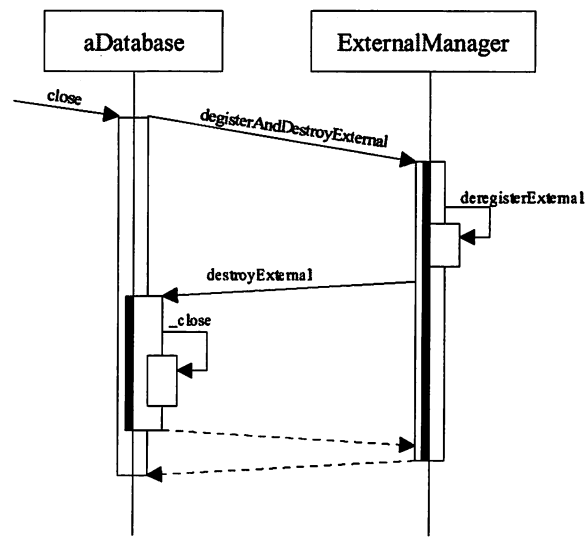


Figure 9: Unregistering from the `ExternalManager`

To avoid deadlocks at stabilization, the order in which the *monitors* (object locks engaged by the `synchronized` keyword) of the `External` and the `ExternalManager` are acquired is crucial. In the examples, a thick lifeline denotes a locked monitor, a thin one a free monitor.

The rationale for the stabilization protocol is as follows:

1. While a mutator thread is performing work on an `External`, its could potentially be in an inconsistent state. This prevents us from simply stopping all mutator threads before stabilization, i.e. stabilization must be performed in the running system.

2. After external state has been internalized, no mutator thread must be allowed to manipulate the `External`. Consequently, the stabilizing thread must hold on the `External`'s monitor after internalization.

3. No `External` may be registered or unregistered during stabilization. [2]

This leads to the following protocol: when the Spotless store is stabilized (which is triggered by the user changing to another application), the `ExternalManager` is first locked. After that, it gives all registered `Externals` the chance to internalize their external state in the reverse order to that in which they registered with the `ExternalManager`. This is performed by (i) acquiring the `External`'s monitor and (ii) sending the `External` the `internalizeExternal` message. At stabilization time, all `Externals`' monitors are held.

On application resumption the process proceeds in the reverse order. The `ExternalManager` will (iii) send the `External` the `recreateExternal` message and will then (iv) release the object's monitor. The recreation messages are sent in the same order in which the `createExternal` methods were once invoked.

To avoid deadlocks involving a stabilizing thread it is necessary that the `ExternalManager` object is always locked *before* the `External`. Note that the `new`, `_open` and `_close` methods in the examples are not synchronized. In general, no method that calls `createAndRegisterExternal` or `unregisterAndDestroyExternal` may synchronize on the `External` or else a deadlock is possible.

Figure 10 illustrates the stabilization process with two registered `Externals`. Note that the thick line representing the lock on the `External` is actually starting before the subsequent invocation of the `stabilizeStore` method and is held until the `ExternalManager` sends the `recreateExternal`

[2] This is for simplification. A more complicated scheme interleaving internalization and (un-)registration could possibly be thought of.

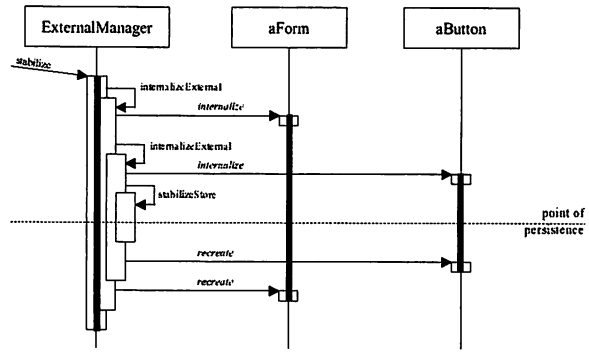


Figure 10: Performing the `ExternalManager.stabilize` Operation

call. Also note the use of persistent threads: the stabilizing thread is simply frozen by calling `stabilizeStore`; it holds on to its locks until it is resumed and then continues execution.

Recapping this section we observe that the `External` protocol offers a thread-safe way to manage a persistent application's external data. It provides an atomic registration/unregistration mechanism and is instrumental in preventing deadlocks and the persistence of inconsistent state.

4 Remaining Safety Issues

By disabling PalmOS' write protection as described in section 3, we create a new safety problem: it is possible for a VM to overwrite the memory of dormant applications and cause damage to them. This would need to be addressed in case our contributions were made into a product.

One could argue that a correctly implemented VM would never attempt any access outside its own store. Hence, debugging the VM and all added native code would "asymptotically" eliminate the problem. However, if one wants to be on the safe side, some form of hardware write protection is required.

If we accepted the limited amount of primary memory in the original Spotless VM, there would be a simple solution. We might then have implemented the persistence of execution state in dynamic RAM by copying all of it into a database record on shutdown and read it back into dynamic RAM on resumption. However, we do not have any intention

to claim: "64 K should be enough for everybody."

It should be possible to protect only part of the static RAM, *not all* of it. One could then arrange during each switch between applications that all passive programs are protected and that only records of the designated active application are exposed to unprotected write access. For example, if the hardware protection is governed by an address limitation register, then it would be straightforward to relocate each application so that only the running one resides in dynamic RAM.

The remaining problem is that memory *outside* records could be altered inadvertently, which would compromise the integrity of meta data (e.g., handle collections, database indices) managed by the OS. Ideally, the OS would place meta data in the protected memory range. In case it does not, a promising approach would be to save all of the meta data occurring in the unprotected memory area to a database before they can be altered and to restore them before yielding control to another program or the OS.

5 Related Work

Pocket Smalltalk [Arsenau and Brault, 1999] is a Smalltalk-80 implementation for the Palm Connected Organizer. Like our system, Pocket Smalltalk circumvents PalmOS's database API and implements an in-store execution model by directly modifying its segmented heap in the static RAM area. The system does not offer an elaborate protocol to manage the persistent state of external data.

MERPATI [Suezawa, 2000] extended the JDK 1.1 VM through a checkpointing mechanism for purposes of error recovery and migration of computations. After suspension of all computation, the whole program state is transferred to disk. Contrary to our work, MERPATI does not offer a checkpointing protocol to internalize partially external state. Also, the system does not persist class file contents but requires them to be installed at every migration target. MERPATI employs a conservative garbage collector, so type accurate descriptions of stack contents are only needed for checkpoints. Consequently, the system computes reference maps on demand, instead of updating them with every bytecode instruction, as our system does.

The PJama research prototype [Atkinson and Jordan, 2000] features a modified high-performance Java 2 VM and runs on top of a recoverable, scalable persistent object store. Through incremental faulting and eviction on a per-object basis, the system can operate on stores much larger than physical memory. PJama does not persist threads — this allows for a simpler checkpointing procedure, as synchronization issues between the checkpointing thread and other mutators do not need to be addressed. However, in order to achieve a consistent checkpoint, all executing threads have to agree that they have currently reached a consistent state by using synchronization on the application level.

The GemStone/J^{TM} Persistent Cache Architecture^{TM} [GemStone, 2000] uses modified Java 2 VMs to provide access to a multi-user, transactional store. Its multi-user architecture provides a logical separation between the store and the execution engines operating on it. Hence, persistence of execution state (threads) is not within the scope of this system.

6 Conclusion

We extended the functionality of the Spotless VM with automatic memory management that provides orthogonal persistence, including thread state. Java programs running on the Spotless VM are continuous processes: they can be suspended and then later resumed at exactly the same point of execution as where they left off. They may even be resumed on a different device, since suspended programs can simply be beamed between Palm organizers.

Given the described enhancements to the Spotless VM, it is no longer necessary for the programmer to use a Palm database for making data persistent. Palm databases are only needed to share data among applications.

There is one exceptional situation in which programmers do need to write persistence-related code: to maintain external resources that are not under the control of the Spotless runtime system. For such cases, we provide a callback API to include these resources in the suspension/resumption phases. Leveraging the automatic persistence of threads and synchronization primitive, such call-

backs can be scheduled and executed without compromising data consistency.

Generally, external resources should be dealt with at the library level, such that the *application* programmer is completely unburdened from the task of achieving persistence.

Our main approach is to represent program images directly in stable memory (battery-backed RAM). Because the OS can move memory blocks representing dormant programs around, we build on top of a compacting GC to relocate pointers on program resumption.

Compared with the previous Spotless VM, there is no loss in execution speed, no increase in application footprint and only a small increase in VM footprint (about 4KB or 5%).

7 Acknowledgments

The original Spotless VM was designed and built by Antero Taivalsaari, Bill Bush and Doug Simon.

The exact garbage collection was implemented by Matthew Seidl during his summer internship supervised by Mario Wolczko.

We want to thank Malcolm Atkinson, Mario Wolczko, Mick Jordan and Grzegorz Czajkowski for their many helpful suggestions to improve this document.

References

[Arsenau and Brault, 1999] Arsenau, E. and Brault, A. (1999). Pocket Smalltalk & Pocket Java. Presentation at 1999 Smalltalk Solutions Conference, http://www.pocketsmalltalk.com/Solutions99/Final.PPT.

[Atkinson and Jordan, 2000] Atkinson, M. and Jordan, M. (2000). A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform. Sun Labs Technical Report TR-2000-90, Sun Microsystems Laboratories.

[Gawecki and Wienberg, 1998] Gawecki, A. and Wienberg, A. (1998). Report on the Tycoon-2 Programming Language. Technical Report Version 1.0, Higher-Order GmbH.

[GemStone, 2000] GemStone (2000). GemStone/J. GemStone Systems, http://www.gemstone.com.

[Haddon and Waite, 1967] Haddon, B. K. and Waite, W. M. (1967). A Compaction Procedure for Variable Length Storage Elements. *Computer Journal*, 10:162–165.

[Jones and Lins, 1996] Jones, R. and Lins, R. (1996). *Garbage Collection Algorithms for Automatic Dynamic Memory Mangement*. John Wiley & Sons.

[Jordan and Atkinson, 1999] Jordan, M. and Atkinson, M. (1999). Orthogonal Persistence for the Java™ Platform - Rationale. in preparation.

[Kornacker, 1995] Kornacker, M. (1995). Persistent Savepoints for Long-Lived Activities in Open Environments (in German). Master Thesis, Fachbereich Informatik, Universität Hamburg.

[Mathiske et al., 1995] Mathiske, B., Matthes, F., and Schmidt, J. W. (1995). On Migrating Threads. In *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel.* Also appeared as TR FIDE/95/136 , FIDE Technical Report Series , FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ.

[Mathiske et al., 1997] Mathiske, B., Matthes, F., and Schmidt, J. W. (1997). On Migrating Threads. *Journal of Intelligent Information Systems*, 8(2):167–191.

[Suezawa, 2000] Suezawa, T. (2000). Persistent Execution State of a Java Virtual Machine. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 160–167, San Francisco, CA.

[Sun, 1999] Sun (1999). The K Virtual Machine (KVM) - A White Paper. Technical report, Sun Microsystems, Inc.

[Taivalsaari et al., 1999] Taivalsaari, A., Bush, B., and Simon, D. (1999). The Spotless System: Implementing a Java™ System for the Palm Connected Organizer. Sun Labs Technical Report TR-99-73, Sun Microsystems Laboratories.

[Weikard, 1998] Weikard, M. (1998). Design and Implementation of a Portable Multiprocessor Capable Virtual Machine for a Persistent Object Oriented Programming Language. Master's thesis, Department of Computing Science, University of Hamburg, Germany. in German.

Energy Behavior of Java Applications from the Memory Perspective*

N. Vijaykrishnan, M. Kandemir, S. Kim,
S. Tomar, A. Sivasubramaniam and M. J. Irwin
Dept. of Computer Science and Engg.
The Pennsylvania State University
University Park, PA 16802
vijay@cse.psu.edu

Abstract

With the anticipated dramatic growth of computing devices for mobile and embedded environments, energy conscious hardware and software design has taken center-stage together with performance. At the same time, there is an increasing need to provide a portable and seamless software environment for application development and interoperability. This paper takes an important step in the confluence of these two emerging trends, by examining the energy behavior of the memory system in the execution of Java applications. It is crucial to understand and optimize the energy behavior of the memory system since instructions referencing memory can contribute to a large fraction of the energy consumption when executing Java applications.

Using an off-the-shelf JVM, a validated memory energy model, and a detailed simulator, this paper presents a characterization of the energy consumption by the cache and main memory when executing the SPEC JVM98 benchmarks in the JIT and interpreter modes. The energy consumption is profiled for the different hardware components (instruction cache, data cache, memory) and software components (class loading, garbage collection, dynamic compilation). The results from such a characterization are useful to the hardware designer for cache organizations and architectural enhancements for reducing energy consumption. They are also useful to the application and runtime system designer to identify energy bottlenecks, and for code restructuring or algorithm redesign to alleviate these bottlenecks.

*This research is supported in part by grants from NSF CCR-0073419, Pittsburgh Digital Greenhouse and Sun Microsystems.

1 Introduction

Computing is becoming a pervasive and ubiquitous part of everyday life. This has led to important design considerations from the software, the need to provide a seamless and portable software platform that facilitates easy inter-operability, and hardware, the need to incorporate energy and form factor conscious designs and to reduce time-to-market and cost, angles. This paper brings these two design considerations together by examining the energy consumption of JVM implementations that form the cornerstone of Java [1], which is one of the software platforms for the seamless integration of diverse ubiquitous/embedded devices. In particular, this paper focuses on the energy consumed by the memory system when executing the SPEC JVM98 benchmarks [27].

Java provides portability across systems by specifying only the format and semantics of the bytecodes, without placing any restrictions on how they are executed. Consequently, the choice of the JVM implementation style depends on the performance, availability of hardware resources, as well as energy criteria. Previous studies have mainly looked at performance and hardware resource issues between JVM implementation styles. This is one of the first studies to examine JVM implementation styles from the energy viewpoint. Energy, measured in Joules, is the power consumed over time and is an important metric to optimize for prolonging battery life. The importance of optimizing for this metric has been further accentuated by the slow improvements in the energy capacity of batteries.

The focus of this paper is in studying

the energy consumption behavior of the Java codes from both software and hardware perspectives. While energy consumption is an important issue in mobile systems of varying degree of resource constraints, the experimental setup and evaluation benchmarks used in this work are representative of high-end mobile devices such as laptops. The JVM used in our experiments is the Sun Labs Virtual Machine for Research, formerly known as ExactVM (EVM) [31]. While current low-end mobile devices use small footprint JVMs that use simple interpretation, more sophisticated JVMs such as the one used in this work are attractive, due to their performance, for high-end mobile devices such as laptops and emerging low-end mobile devices such as palmtops with large memory modules.

We utilize the SPEC JVM98 benchmarks in this work and, specifically, focus on characterizing the memory system energy consumption for the following reasons:

- It has been observed [37, 4] that the memory system can consume a large fraction of the overall system energy, making this a ripe candidate for software and hardware optimizations.
- Among the different instructions executed by the SPARC architecture while executing the SPEC JVM98 benchmarks, the load/store instructions are the primary instructions that access the memory for their operands. On the average, we find the energy consumed by load/store instructions accounts for 58.7% and 52.2% of the total energy in the interpreted and JIT-compiled modes, respectively, when executing the SPEC JVM98 benchmarks (see left side of Figure 1). The energy consumed by load/store instructions is significant considering that on an average, these instructions are observed to constitute only 22.7% and 19.4% of the total instructions for the interpreted and JIT modes, respectively (see right side of Figure 1).
- Finally, Java executions are expected to stress the memory system more than traditional programs [17, 3, 9]. Bytecodes are treated as data, and need to be fetched from memory for interpretation or JIT-compilation and installed. Further, JVM features such as garbage collection make Java executions much more memory-intensive than normal programs.

We examine the energy consumption of the memory system when executing Java programs, with different JVM implementation styles, to understand the hardware and software bottlenecks. From the hardware viewpoint, such information can be used to suggest cache organizations and strategies for optimizing energy-delay criteria. Identifying energy bottlenecks for different software components of the execution, such as class loading, garbage collection or dynamic compilation, can lead to the design of better algorithms and mechanisms to reduce the energy demands. For instance, if garbage collection turns out to be energy hungry, one could either opt to invoke the collector less frequently or design more energy-conscious garbage collection algorithms. The information can also be used to decide when to interpret and when to compile, rather than base this decision purely on performance considerations.

To our knowledge, there has been only one prior study that has attempted to profile the energy consumption of Java programs [10]. However, their use of an actual pocket-computer to measure the current for calculating energy has limited their extent of profiling. It does not provide adequate information as to what hardware components are consuming the energy during the different phases of execution. On the other hand, the use of a detailed simulator helps us profile the energy consumption from both the hardware and software angles to isolate the fraction consumed by the memory system. Using the SPEC JVM98 benchmarks on off-the-shelf JVM implementations in the interpreted and JIT-compiled modes, this paper sets out to answer the following questions:

- How much energy is consumed by the memory system in the execution of Java programs? What fraction is consumed by the cache and what fraction by the main memory? What is the energy breakdown between instruction and data references?
- How do different cache configurations affect the energy consumption? In particular, what is the effect of the cache size and associativity? While cache configurations that traditionally favor lo-

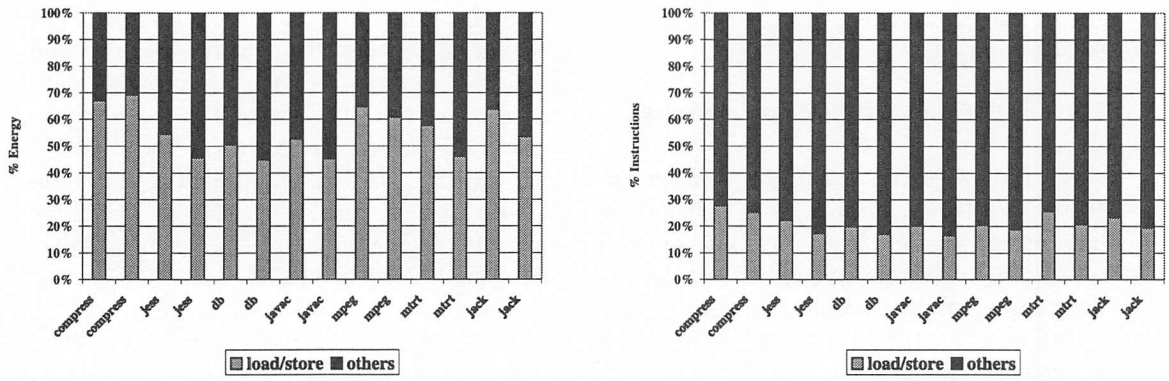


Figure 1: *Energy consumption distribution between load/store and other instructions (left). Distribution of dynamic instructions categorized as load/store and other instructions when executing on a SPARC architecture (right). The energy consumption for memory references are assumed to hit in the cache. Energy consumption numbers for each group of instructions are applied from [34]. Energy consumed in both datapath and memory is included.*

cality would reduce the references to main memory and reduce memory energy consumption, those configurations would also increase the energy cost per access of the cache. Studying these trade-offs would help decide on a good operating point.

- How do these results differ between the two JVM implementation styles — interpretation and JIT compilation? How much energy is consumed by different components of the JVM such as class loader, garbage collector? How do the cache parameters affect these components? What is the impact of application parameters, problem size in particular, on the energy consumption profile? For instance, a larger problem can result in the garbage collector being invoked more often.

2 Experimental Framework

In this section, we first give details on the specific JVM used in our simulations. We then describe the characteristics of the benchmarks used in our experiments, and the simulator used to gather memory access information. The energy model used for the memory system is also described.

2.1 Java Virtual Machine (JVM)

The JVM used in our experiments is the Sun Labs Virtual Machine for Research, EVM [31]. This is a high performance VM that has been designed to facilitate experiments in memory management. It is designed to provide a fast memory system, fast synchronization, and a fast Just-in-Time (JIT) compiler. It is also an adaptive VM, i.e., it detects and accelerates performance-critical or often-used code. Rather than compiling the entire program when it starts, as is done in virtual machines with pure JIT compilers, the EVM starts off running the application using an interpreter. At the same time, it gathers *profiling information* regarding the runtime characteristics of the application and uses this information to dynamically compile certain methods. In all our experiments, methods with loops are compiled at the first invocation itself, and methods without loops are compiled when the invocation count reaches 15, a value determined after extensive experimentation [22].

We identify three parts within the EVM, apart from the actual application execution, where the principal events of interest that stress the memory subsystem occur. The first is *class loading,* when the binary form of a class is brought into the virtual machine on the first invocation, incurring many compulsory cache misses. The second is *dynamic method compilation,* when a method is converted into native code, and the native code is installed in memory for subsequent execution. This event occurs only in the EVM executing as an adaptive compiler. Finally, there is *garbage collection,* which runs periodically as a separate thread and reclaims memory from non-referenced objects. A default heap size of 24 megabytes was used in the experiments. The remaining parts have been grouped together with the execution of native code and

includes object accesses, thread creation, synchronization, etc.

2.2 SPEC JVM98 Benchmarks

We use all seven applications from the SPEC JVM98 benchmark suite [27] for our experiments that are briefly described in Figure 2. The benchmark programs can be run using three different problem sizes, which are named as s100, s10 and s1. While the problem sizes are larger for the s100 dataset size, the sizes do not scale as designated by the labels 100, 10 and 1. In interest of simulation time, especially, the interpreted mode in s100 mode, we use the smaller problem size (`s1`). We believe that the s1 data set is representative of shorter running applications or applets. While some of the results from the s1 observations would be applicable to the larger data sets, it must be noted that the impact of garbage collection and dynamic compilation impacts tend to change for the larger data sets. Thus, we provide s10 and s100 energy breakdowns in Figure 12 to emphasize this difference.

2.3 Memory System Energy Model

For obtaining detailed profiles, we have customized an energy memory simulator and analyzer using the Shade [6] tool-set. Our memory simulator models on-chip instruction cache (Icache), on-chip data cache (Dcache), and off-chip memory, and allows the user to modulate the various parameters for these components. We characterize the overall energy of the memory system by the energy consumed by five components: the instruction cache, the data cache, the buses, the I/O pads and the main memory.

Note that we focus only on the dynamic energy consumption, since in current technology, the dynamic energy accounts for 80% of the total energy whereas the rest, short circuit and leakage, takes only 20% [14]. The energy consumed by the Icache and by the Dcache are evaluated using an analytical model that has been validated to be highly accurate, within 2.4% error, for conventional cache systems [16, 25]. The energy consumed by the caches are based on technology related parameters for 0.8 micron technology. The energy consumption depends on the number of cache bit lines, word lines, and the number of accesses due to both hits and misses. The details of the model are quite involved and can be found in [22]. The address and data buses between the Icache/Dcache and the datapath account for the energy consumed by the on-chip buses. The bus energy consumption is evaluated by monitoring the switching activity on each of the bus lines using a capacitive load of 0.5pF per line [39]. The energy consumed by the I/O pads and the external buses to the main memory from the caches is evaluated similarly for a capacitive load of 20pF per line. The main memory energy is based on the model in [25] and uses a *per main memory access* energy, referred as E_m in the rest of the paper of 4.95×10^{-9} Joules. A 32 megabyte main memory recommended as smallest memory size for running SPEC JVM98 benchmarks [27] is used in this work. The energy consumed by main memory accesses is further broken down into that consumed due to instruction accesses (Imemory) and data accesses (Dmemory).

3 Energy Behavior

In this section, we present a detailed energy analysis of the SPEC JVM98 benchmarks. We first present the overall energy picture. Then, we zoom in on two codes, `javac` and `db`, and investigate their energy behavior by modifying different cache parameters such as cache size and associativity. These experiments give us the *hardware view* of energy consumption; i.e., they help us answer the question of which parts of the memory system consume the most energy. Subsequently, we present an energy breakdown for the benchmarks from the *software viewpoint.* This helps us understand which software components consume the most energy. Experiments are conducted for both JVM implementation styles: *interpreter mode* and *adaptive dynamic compilation mode* (*JIT mode*).

3.1 Overall Energy Picture

Figure 3 shows the percentage energy breakdown for the interpreter and the JIT modes assuming 32 KB two-way set associative instruction and data caches. We see from the figure on the left that most of the energy in the JIT mode is due to memory accesses for data (Dmemory). In contrast, the memory energy due to native SPARC instruction accesses is lower. This is because the in-

Benchmark	Brief Description
`compress`	A high-performance application to compress/uncompress large files; based on the Lempel-Ziv method (LZW)
`jess`	A Java expert shell system based on NASA's CLIPS expert shell system
`db`	A small database management program that performs several database functions on a memory-resident database
`javac`	JDK 1.0.2 Java compiler
`mpeg`	MPEG-3 audio file compression application
`mtrt`	Dual-threaded ray tracer; the only multi-threaded application in the suite
`jack`	A Java parser generator with lexical analyzers; an early version of what is now called JavaCC

Figure 2: *SPEC JVM98 benchmark codes used in the experiments.*

struction accesses exhibit much better data locality and cause less number of off-chip accesses. The same trend is also observed with the interpreter mode of operation as shown in the graph on the right. The instruction accesses in the interpreter mode have better locality than the JIT mode [23], and as a result, the percentage of the instruction cache energy is higher in the former, especially for `compress`. As an example, in the interpreter mode, instruction cache energy consumption for the `javac` benchmark is 12.3% of the overall memory system energy, whereas the corresponding figure in the JIT mode is 8.7%.

The energy consumption for a given unit is directly related to the number of accesses and per access energy cost [37]. It can be observed from Figure 4 that the interpreter consumes significantly more energy than in the JIT mode. A detailed breakdown of energy is shown in Figure 5. This is due to the fact that both the number of instructions and number of data accesses in the interpreter mode are higher than the JIT mode. For example, `javac` consumes 2.15 times more energy in the interpreter mode as compared to the JIT mode. Thus, the use of JIT compilers will be more beneficial not only in terms of performance (as is well-known [23]) but also from the energy viewpoint.

3.2 Hardware View – Impact of Cache Configuration

To investigate the influence of cache configurations on the energy consumption, we conducted experiments with different cache sizes and associativities. It is important to note that the energy consumption in the memory system is highly dependent on the number of cache misses as the off-chip memory energy cost is an order of magnitude larger than on-chip cache energy cost [37]. However, reducing the number of cache misses is normally achieved using caches of larger sizes or associativities that in turn add to the per access energy cost for the cache. Thus, it is essential

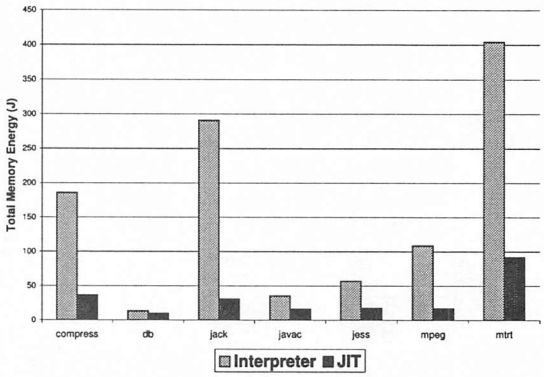


Figure 4: *A comparison of energy consumption when executing in JIT compile mode and interpreter mode. The Icache and Dcache are 32 KB two-way set associative and have 32 byte block size. s1 dataset was used.*

to understand the energy trade-off between better cache locality and increased per access cost.

Figure 6 presents the energy spent in caches and memory for different cache configurations for `javac` and `db`. From these graphs, we can make the following observations. First, when the cache size is increased, we observe a decrease in the overall energy consumption up to a certain cache size. However, beyond a point the working set for instruction or data are contained in the cache, and a size larger than this does not help in improving the locality but does increase the per access cost due to the larger address decoder and larger capacitive load on the cache bit lines. A similar trend is also observed when we change the associativity for a fixed cache size, where increasing the associativity aggressively brings diminishing returns in energy saving. This effect is due to more complicated tag-matching hardware required to support higher associativities. Second, we observe that the instruction accesses seem to take advantage of larger caches better than data accesses. For example, in `javac` with the interpreter mode, when we move from a 4K direct-mapped cache to a 128K direct-

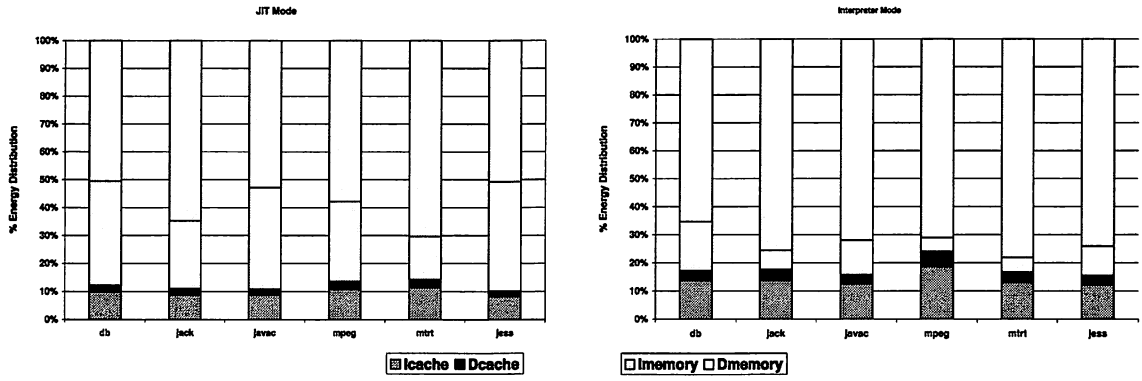


Figure 3: *Energy distribution for the JIT mode (left) and interpreter mode (right). The Icache and Dcache are 32 KB two-way set associative and have 32 byte block size. s1 dataset was used.*

Benchmark	Interpreter Mode								
	Icache		Dcache		Imemory		Dmemory		Total
	Acc.	Eng.	Acc.	Eng.	Acc.	Eng.	Acc.	Eng.	Eng.
compress	21,340.00	98.17	7211.6	33.17	0.6	0.77	42.4	53.71	185.82
db	450.5	1.82	118.6	0.48	1.8	2.34	6.9	8.74	13.38
jack	9,834.9	39.82	2,814.4	11.39	15.6	19.81	173.0	220.05	291.07
javac	1,076.3	4.36	301.1	1.22	3.4	4.34	20.0	25.44	35.36
jess	1,698.3	6.88	477.4	1.93	4.6	5.89	33.1	42.16	56.86
mpeg	4,730.6	19.15	1,432.3	5.80	4.0	5.06	58.1	73.87	108.88
mtrt	12,935.4	52.37	3,657.7	14.81	16.7	21.21	248.3	315.90	404.29
Benchmark	JIT Mode								
	Icache		Dcache		Imemory		Dmemory		Total
	Acc.	Eng.	Acc.	Eng.	Acc.	Eng.	Acc.	Eng.	Eng.
compress	922.4	4.24	330.9	1.52	1.8	2.22	22.2	28.11	36.09
db	238.2	0.97	59.5	0.24	2.9	3.70	3.9	5.00	9.91
jack	657.1	2.66	168.4	0.68	5.8	7.4	15.5	19.7	30.44
javac	353.1	1.43	88.4	0.36	4.7	5.95	6.8	8.68	16.42
jess	352.7	1.43	89.6	0.36	5.4	6.88	7.1	8.95	17.62
mpeg	456.0	1.85	129.0	0.52	3.9	4.94	7.9	10.02	17.33
mtrt	2,535.6	10.27	698.7	2.83	11.1	14.05	50.8	64.43	91.58

Figure 5: *Number of cache and memory accesses (denoted* **Acc.***) in millions and absolute energy values (denoted* **Eng.***) in Joules in interpreter and JIT mode.*

mapped cache, the instruction memory energy drops from 132.8J to 17.0J. On the other hand, except for the move from 4K to 8K, the data memory energy does not vary significantly across different cache configurations for the dataset size that is used in these experiments (s1). Finally, as far as the general energy trend is concerned, the JIT mode behaves similar to the interpreter mode except for the fact that the actual energy values are much smaller, less than half typically, and in some cases the cache configuration that results in the minimum energy consumption is different in the JIT mode from that of the interpreter mode.

It should be noted that although the number of memory accesses in the interpreter mode is higher than the JIT mode, the memory footprint of the former is smaller [23]. The increase in memory footprint for JIT compiler can be due to the additional libraries required for the JIT optimizations and dynamic code installation. For example, the SPARC and Intel versions of the JIT compiler proposed in [7] themselves require 176Kbytes and 120Kbytes. The influence of extra space required for compiled code in JIT mode is found to require 24% more memory space as compared to interpreter mode for the SPEC JVM98 benchmarks, on the average [23]. Consequently, in embedded environments where we have the flexibility of designing custom memory for a given application code [5, 2], we can potentially use a smaller memory for the interpreter mode of operation. In order to capture the effects of lower memory overheads due to the absence of dynamic compilation overheads, we scale the memory size of the interpreter relative to that of the JIT compilation mode. It must be noted that a smaller physical memory will incur less energy due to smaller decoders and less capacitive loading on the bitlines. We will assume that the energy cost of a memory

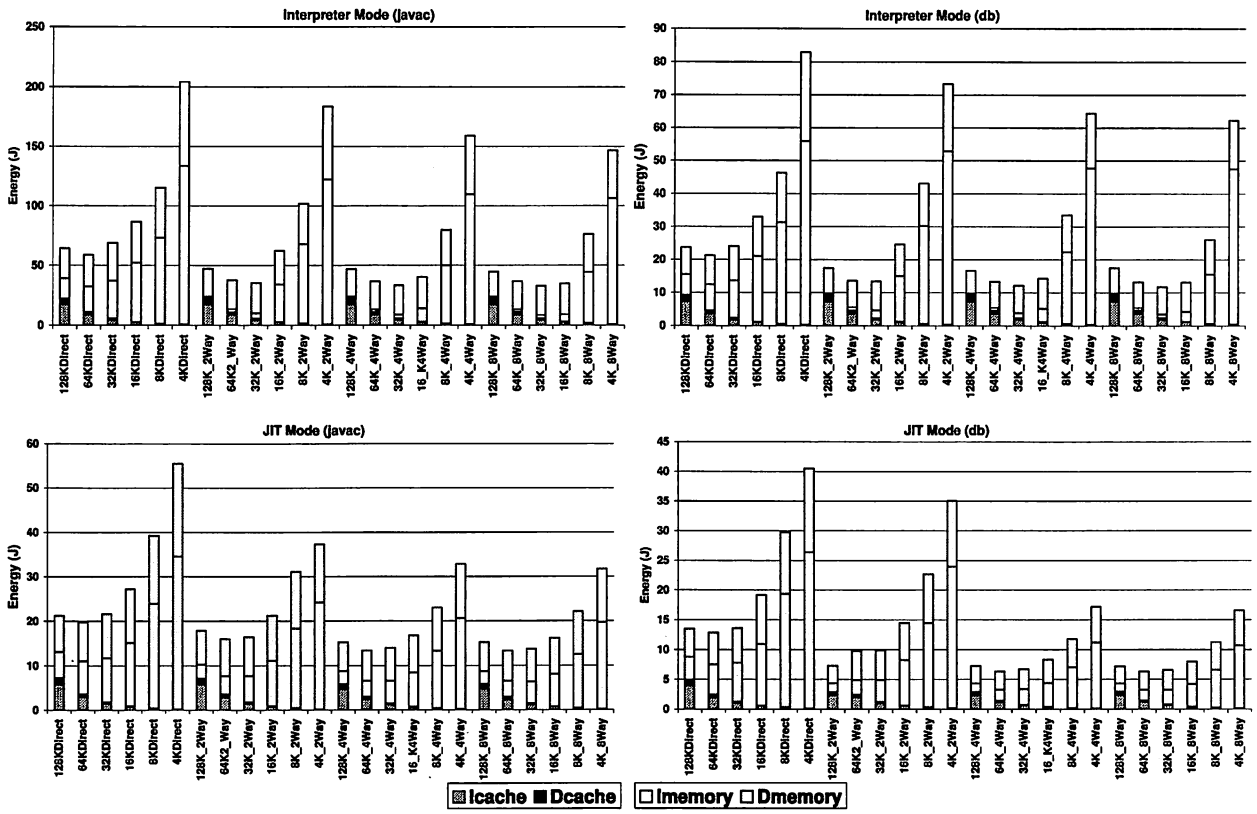


Figure 6: *Energy consumption (memory system) of* `javac` *and* `db` *for different cache configurations. The cache sizes and associativities on the x-axis are for both the instruction and data caches. s1 dataset was used.*

access decreases linearly with memory size for the purposes of this comparison.

Figure 7 gives the total energy consumptions for `db` for different ratios of memory footprint between the interpreter and JIT compiler. For the purposes of this approximation, we have neglected the effect of increased garbage collection overhead which would result when reducing the memory size. The way to interpret the graph is as follows. In `db`, the memory overhead of the JIT mode needs to be at least 1.67 (1/.6) times, (corresponding to 0.6 scaling factor) more than that of the interpreter mode before the interpreter becomes preferable from the energy viewpoint. Until then, JIT is preferable. The observed expansion in data segment for the JIT compilation mode is limited to 24% on an average [23] and the overhead of current JIT compilers is much smaller than the heap size (24 megabytes) needed for both modes. Hence, while one might think that reducing the memory size makes interpretation more attractive, the above observations show that the size expansion in JIT compilation mode is not significant enough to influence optimal energy consumption choice, even neglecting the increased GC overhead. However, if this footprint expansion becomes too large due to some JIT optimizations [38, 28, 21] that increase code size (e.g., in-lining), or the compiler becomes much larger compared to other resources such as heap space required in both modes one may need to re-evaluate this trade-off and select the suitable execution mode during memory system construction for an embedded device where physical memory space is limited.

Main memory has long been a major performance bottleneck and has attracted a lot of attention (e.g., [26]). Changes in process technology have made it possible to embed a DRAM on the same chip as the processor core. Initial results using embedded DRAM (eDRAM) show an order of magnitude reduction in energy cost per access [26]. Also, there have been significant changes in the DRAM interfaces that can potentially reduce the energy cost of external DRAMs. For example, unlike conventional DRAM memory subsystems that have multiple memory modules that are active for servicing data requests, the

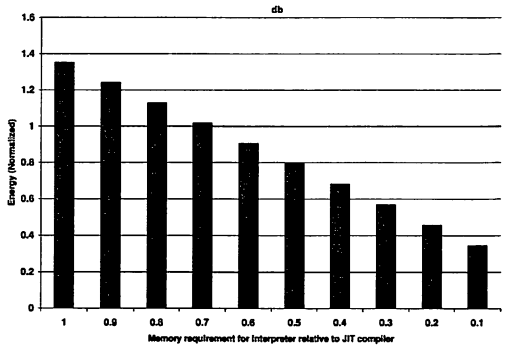


Figure 7: *Relative energy consumption of interpreters as compared to JIT Compiler. The memory size for the interpreter is varied relative to that of the JIT Compiler to capture the differences in the overheads associated with the storage associated with different compilers and the code expansion that can occur during native code generation and installation. An instruction cache and a data cache, both 32 KB, two-way associative with 32 byte block size, are used.*

direct RDRAM memory sub-system delivers the full bandwidth with only one RDRAM module active. Similarly, new technologies such as magnetic RAMs consume less than one hundredth the energy of conventional DRAMs [33]. Also, based on the particular low power operating modes that are supported by memory chips and based on how effectively they are utilized, the average energy cost per access for external DRAMs can be reduced by up to two orders of magnitude [8]. In order to study the influence of these trends, we performed another set of experiments using four different E_m values: 4.95×10^{-9}J (our default value), 2.45×10^{-9}J, 2.45×10^{-10}J, and 4.95×10^{-11}J. Each of these value might represent the per-access cost for a given memory technology. Figure 8 shows the *normalized* energy consumptions (with respect to the interpreter mode with the default E_m value). We observe that the ratio of the total memory energy consumed by the interpreter mode to that of the JIT mode varies between 1.05 (2.07) and 1.80 (2.85) for db (javac) depending on the E_m value used. We also observe that the relative difference between energy consumed in interpreter mode and JIT mode increases as E_m reduces due to better technologies. For example, the energy consumed in JIT compilation mode is half of that consumed in the interpreter mode for most energy-efficient memory while it is around 70% of interpreter energy for most energy consuming configuration when executing db. This indicates that the even when the process technology significantly improves in the future, the JIT will remain the choice of implementors from an energy perspective.

3.3 Software View – Energy Distribution across Software Components

In this part, we profile the energy consumption between different software components such as class loading (load), dynamic compilation (compile), garbage collection (GC), and the rest of the execution (exec). An understanding of this breakdown can provide the Java Virtual Machine (JVM) implementors and application designers with an indication of which components of their software consume the most energy and help them concentrate on optimizations to those particular components. For instance, if the energy cost due to object accesses is high, one could modify the object allocation strategies used by the JVM or apply object reuse strategies at the software level. Similarly, if the garbage collector [15, 29, 9] component is significant, one could opt for an improved algorithm or modulate the frequency of its invocation.

Figure 9 gives the energy distribution for the software components in both the interpreter and JIT modes. For example, jack executing the interpreter mode, the instruction accesses consume 60J and data accesses consume 232J. The corresponding energy numbers for the JIT mode are much lower at 10J and 20J respectively. These results are in consonance with the better locality of instruction accesses in the interpreter mode as discussed earlier. In the interpreter mode almost all the energy is spent in the interpretation and GC and class loading were found to be less than 2% of the overall energy consumption. Although execution takes the largest amount of energy in the JIT mode, the dynamic compilation also consumes a significant amount of energy. This is due to two main reasons [23]. First, there are abrupt changes in the working set during dynamic compilation as the code and data structures used by the compiler are different from that for the rest of the JVM. Thus, when we move to the code generation phase, we experience poor locality in the cache (data and instruction) accesses, and this in turn causes more references to the memory (both Imemory and Dmemory). Second, when code is installed af-

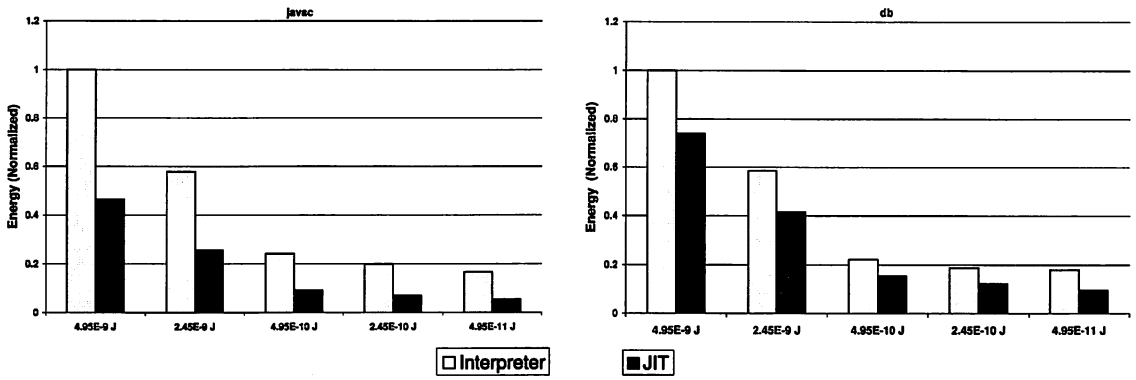


Figure 8: *Overall memory system energy consumption with different values of E_m for the interpreter and the JIT mode.*

ter dynamic compilation, it causes references to main memory. We observe that (in the JIT mode), on an average, the dynamic compilation consumes 24% of the overall energy across the benchmarks. Figure 11(d) breaks down the energy consumption by the hardware components during dynamic compilation and shows that Imemory and Dmemory are responsible for the bulk of the overhead.

In the rest of this discussion, we focus only on the JIT mode since the energy consumption for the interpreter is dominated entirely by the interpretation. Figure 10 gives the energy breakdown of `javac` into different software components with different cache configurations. We observe that (as opposed to class loading and garbage collection) the dynamic compilation and execution can take advantage of larger cache sizes. Data from other experiments [22] show that the energy consumption during loading is mainly dominated by compulsory misses. Hence, the number of total misses during loading is fairly constant across different cache configurations. However, there are small variations in energy consumption with changes in cache configuration as the per-access energy cost is affected not only by number of accesses but also by the energy cost of the tag-matching hardware and the capacitive load of bit lines. As can be observed in class loading profile in Figure 11(a), most of the energy is consumed by the data memory. It should be noted that some Java environments may be running multiple applications concurrently, in which some of the class loading costs can be amortized over the different applications [10].

We see from Figure 9 that the garbage collector consumes a very small fraction of the energy. Its energy consumption due to data accesses is higher than that due to instruction accesses as the garbage collector code itself is very small (i.e., good Icache locality) but the data accessed by the GC has a relatively poor locality. In fact, our detailed analysis shows that most of the energy expended in data memory is a result of the the cache misses. More innovation in improving the data locality of garbage collection will be valuable from energy perspective. While the absolute energy consumed by the garbage collector is small compared to overall execution in these experiments, we believe that the need for more aggressive garbage collection for limited memory embedded systems will make this component more important. It must be noted that the energy consumed in the garbage collection portion is also influenced by the choice of the algorithm and the size of the heap. The size of the heap can influence the number of times the garbage collector is invoked. For example, when we varied the heap size from 24M to 8M, the energy consumed by garbage collection increases eight fold when executing `mtrt` (s100 dataset and JIT compilation mode). The dataset of the application can also influence the energy consumed by the garbage collector. As an example, we found that the GC is responsible for nearly 14% of total data misses for `s100` data set (compared to 7% with `s10`) in the JIT mode, for `javac`, contributing to the overall energy more significantly. More detailed analysis of these tradeoffs in garbage collection energy consumption is beyond the scope of this work and is an interesting area of research in itself.

The execution of compiled code consumes the major chunk of the energy and Figure 11 shows the energy distribution for the different

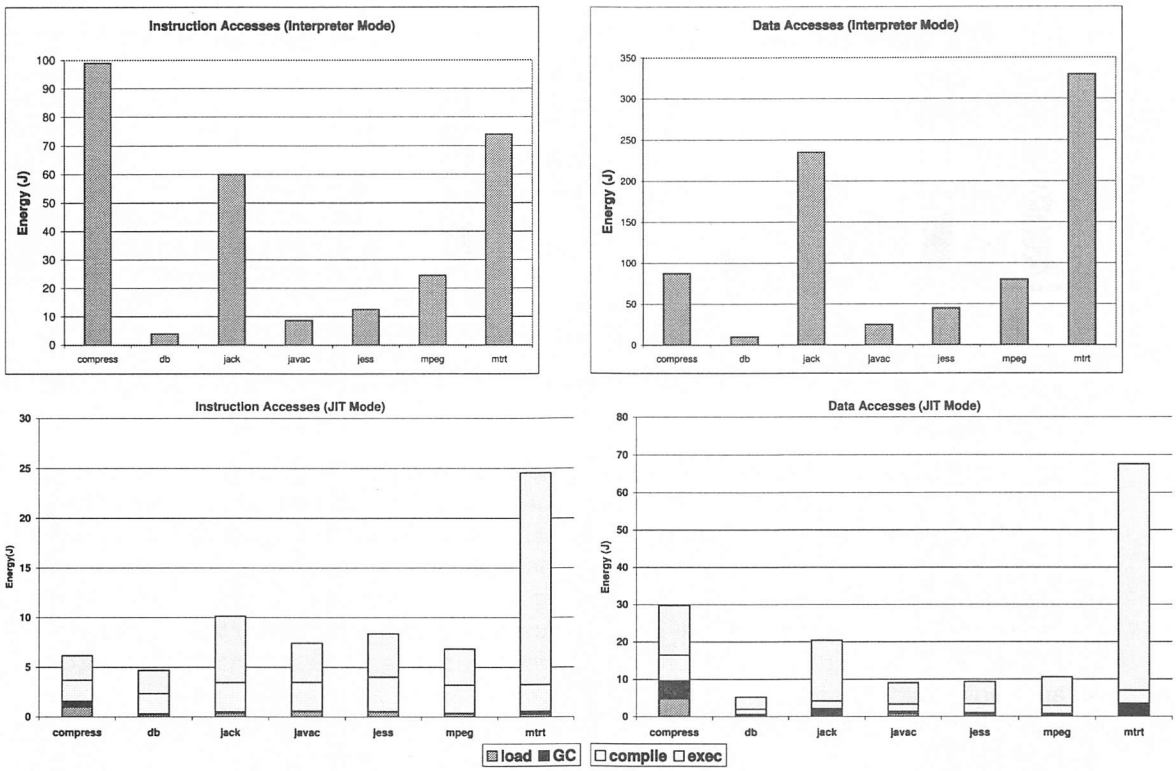


Figure 9: *Energy distribution based on software components. Instruction access energy involves Icache energy and Imemory energy, and data access energy involves Dcache energy and Dmemory energy. In the interpreter mode, the class loading and GC portions were small as compared to the actual interpretation of the code and hence are not shown separately in this breakdown. In the JIT mode, this component is mainly comprised of the energy spent in executing the native code after compilation.*

hardware and software components of the JIT mode. Overall, observing the trends shown in Figure 11, it is interesting to note that different applications in SPEC JVM98 exhibit different energy behaviors. For instance, while `mtrt` consumes the maximum energy during the execution phase, its energy consumption is smaller than that of `compress` during loading, garbage collection, and dynamic compilation. The energy consumption in different software components is a function of the number of classes loaded, the size of the classes, the number of methods compiled, the number of times a method is invoked after compilation, the heap size determining the frequency of GC invocation, the size of data set and the heap allocation, and memory access behavior during execution. Since the actual execution of the compiled code is the dominant component, we need to focus on developing techniques to reduce the energy consumed by this component. Optimizations during the JIT compilation phase (e.g., [28, 38]) can also potentially improve the energy efficiency of the execution phase, sometimes at the cost of increasing energy consumption due to dynamic compilation itself.

Finally, we would like to emphasize that the energy behavior in the different portions of the JVM is also dependent on the dataset size. We observe from Figure 12 that the share of class loading and dynamic compilation are comparatively smaller for the s100 dataset as compared to s10 dataset.

4 Concluding Remarks and Future Work

This paper has taken an important step towards the confluence of two emerging design considerations for ubiquitous and embedded computing: the need for a seamless and portable software platform for easy application design and interoperability, and the need for energy conscious system design. By focusing specifically on the Java runtime system and the SPEC JVM98 benchmarks, this paper has analyzed the energy consumption in the memory system for these appli-

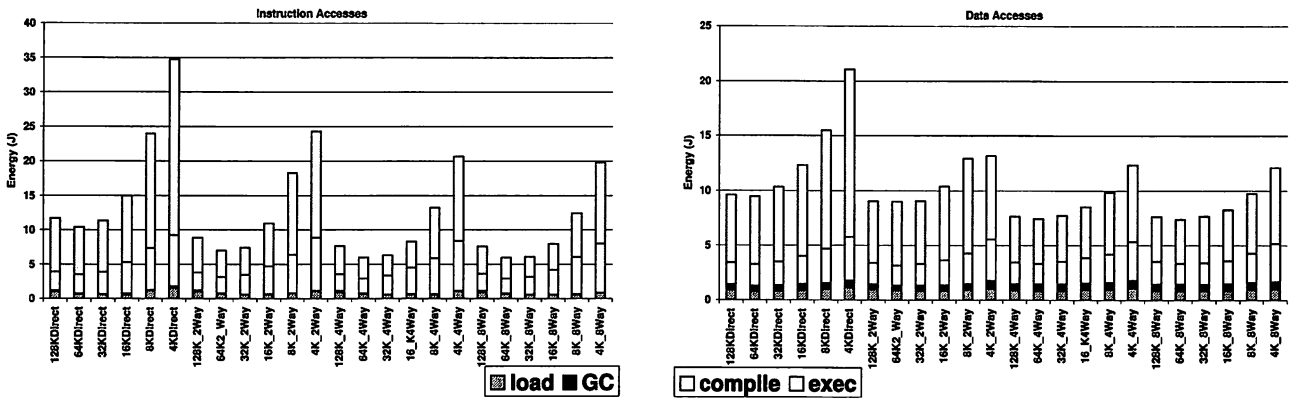


Figure 10: *Energy distribution (based on software components) for* `javac` *with different cache configurations (JIT mode).*

cations. The motivation for this study stems from the observation that instructions accessing the memory system account for over 50% of the energy consumption for these benchmarks. As applications get larger and become more data centric, they are likely to stress the memory system even more.

Using an off-the-shelf JVM, a validated energy model for the memory system, and a detailed simulator, this paper has presented a characterization of the energy consumption in the cache and main memory due to instruction and data references of the SPEC JVM98 benchmarks. The effect of the JVM implementation style (interpretation or JIT compilation) has also been studied, in addition to breaking down the energy consumption between different software components of the JVM — class loading, garbage collection, and dynamic compilation. The detailed profiles from this study can help towards hardware enhancements, in terms of cache and memory organization, and even algorithmic and software designs for energy conscious application and JVM designs.

This study has helped us make the following general observations:

- Main memory energy consumption is more dominant than that of the caches, with the main contributor being data references. Interpretation tends to stress the instruction cache more than JIT compilation, mainly because of the better locality. Overall, from the energy viewpoint, the JIT approach is a better alternative than interpretation.

- Interpretation has been a popular alternative for limited memory systems, since it requires less space than a JIT compiler. Lesser space, smaller memory, also implies a reduction in energy cost per access, which can be another argument one could use in favor of interpretation. However, our study reveals that the saving in energy per access due to smaller memory is not sufficient to compensate for the energy consumed by the larger number of accesses and longer execution time of interpretation compared to JIT.

- Cache organizations that favor locality can decrease the main memory energy consumption, while increasing the cost per cache access. These two contrasting influences make it necessary to decide on a good operating point for cache design that takes both locality and energy into account.

- In the interpreted mode, the energy for the actual interpretation of byte codes clearly dominates any other portion of the JVM. The energy consumed by the dynamic compilation in the JIT mode is quite significant, mainly due to the code installation and subsequent execution misses.

We believe the characterization study in this paper will be helpful for JVM implementors to understand the impact of their decisions on the energy consumption of the system. As this paper is one of the first attempts to characterize energy-behavior of the Java codes, we believe there is lot of scope for enhancements. First, we need to experiment more thoroughly on the effect of data set sizes and with different types of Java applications executing on mobile environments. Second,

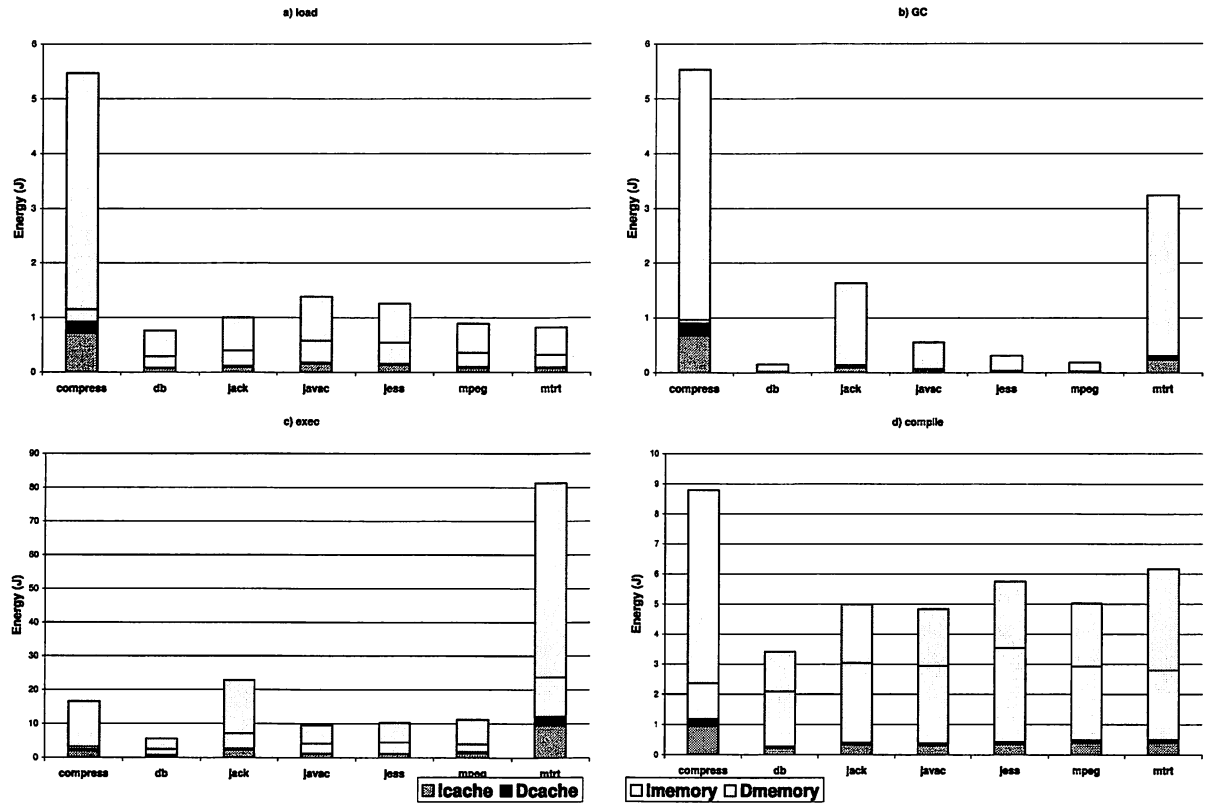


Figure 11: *Energy breakdown for different components. Note that the Y-axis scales for the different graphs are different. The Icache and Dcache are 32 KB two-way set associative and have 32 byte block size.*

we plan to study the impact of technology changes such as increased wire capacitances and leakage power on our study. We plan to address these in our future work.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language,* Addison-Wesley, 1998, Second Edition.

[2] F. Balasa, F. Catthoor, and H. De Man. Exact evaluation of memory area for multi-dimensional processing systems. In Proc. *the IEEE International Conference on Computer Aided Design,* Santa Clara, CA, pages 669–672, November 1993.

[3] B. D. Cahoon, and K. S. McKinley. Tolerating latency by prefetching Java objects. In Proc. *the Workshop on Hardware Support for Objects and Microarchitectures for Java,* October 10, 1999.

[4] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. DeMan. Global communication and memory optimizing transformations for low power signal processing systems. In Proc. *the IEEE Workshop on VLSI Signal Processing,* pages 178-187, 1994.

[5] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. Custom memory management methodology – exploration of memory organization for embedded multimedia system design. *Kluwer Academic Publishers,* June, 1998.

[6] Bob Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In Proc. *ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems,* pages 128–137, May 1994.

[7] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, M. Wolczko. Compiling Java just in time. IEEE Micro, 17(3), pp. 36-43, May-June 1997.

[8] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM energy management using software and hardware directed power mode control. In Proc. *International Conference on High Performance Computer Architecture,* Jan 2000, To appear.

[9] S. Dieckmann and U. Holzle. A Study of the allocation behavior of the SpecJVM98 Java benchmarks. In Proc. *ECOOP'99,* 1999.

[10] J. Flinn, G. Back, J. Anderson, K. Farkas, and D. Grunwald. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In Proc. *International Conference on Measurement and Modeling of Computer Systems,* Santa Clara, California, June 17-21, 2000.

[11] K. Ghose and M. B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers, and bit-line segmentation. In Proc. *1999 International Symposium Low Power Electronics and Design,* 1999, pages 70–75.

[12] R. Gonzales and M. Horowitz. Energy dissipation in general purpose processors. *IEEE Journal of Solid-State Circuits,* 31(9):1277–1283, Sept 1996.

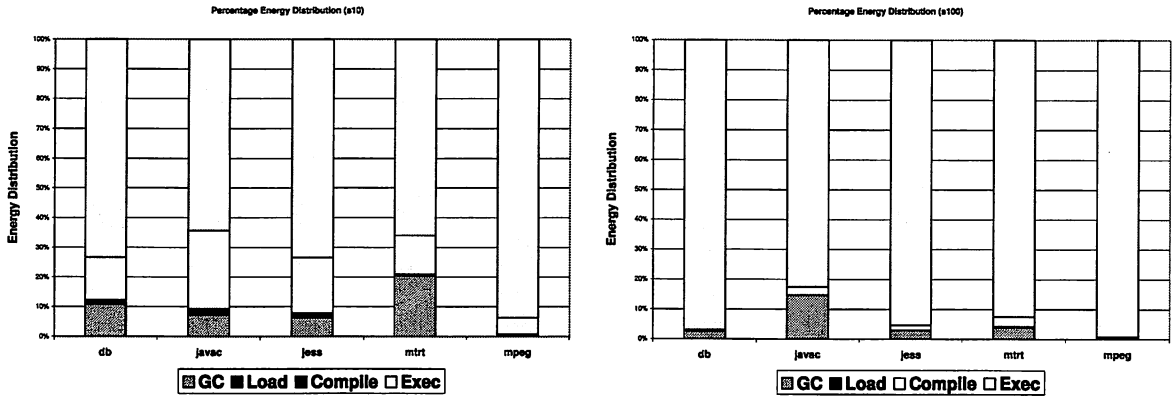


Figure 12: *Energy distribution (based on software components) for five of the benchmarks for (a) s10 (b) s100 dataset. 16K 2-way associative data and instruction cache were used.*

[13] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the desing of the Alpha 21264 microprocessor. In Proc. *the Design Automation Conference,* San Francisco, CA, 1998.

[14] M. J. Irwin and N. Vijaykrishnan. Low-power design: From soup to nuts. Tutorial Notes, *ISCA,* 2000.

[15] R. Jones and R. Lins. *Garbage Collection Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1996.

[16] M. B. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In Proc. *International Symposium on Low Power Electronics and Design,* pages 143–148, 1997.

[17] Jin-Soo Kim and Yarsun Hsu. Memory system behavior of Java programs: Methodology and analysis. In Proc. *International Conference on Measurement and Modeling of Computer Systems,* Santa Clara, California, June 17-21, 2000.

[18] A. Krall. Efficient Java VM Just-in-Time compilation. In Proc. *Parallel Architectures and Compilation Techniques,* Paris, France, 1998.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification,* Addison Wesley, 1997.

[20] H. McGhan and M. O'Connor. PicoJava: A direct execution engine for Java bytecode. *IEEE Computer,* pp. 22–30, October 1998.

[21] J. Moreira, S. Midkiff, M. Gupta, P. Artigas, M. Snir, and R. Lawrence. Java Programming for High-Performance Numerical Computing. *IBM Systems Journal,* 39(2), 2000.

[22] Anupama Murthy. *Memory System Characterization of Java Applications,* Masters Thesis, Dept. of Computer Science and Engineering, The Pennsylvania State University, May 2000.

[23] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, and A. Sivasubramaniam. Architectural issues in Java run-time systems. In Proc. *International Conference on High Performance Computer Architecture,* pages 387-398, January 2000.

[24] K. Roy and M. C. Johnson. Software design for low power. *Low Power Design in Deep Submicron Electronics,* Kluwer Academic Press, October 1996, Edt. J. Mermet and W. Nebel, pp. 433–459.

[25] W-T. Shiue and C. Chakrabarti, Memory exploration for low power, embedded systems, CLPE-TR-9-1999-20, Technical Report, Center for Low Power Electronics, Arizona State University, 1999.

[26] P. Song. Embedded DRAM finds growing niche. *Microprocessor Report,* pages 19–23, August 4, 1997.

[27] *SpecJVM98 Benchmarks.* http://www.spec.org/osg/jvm98

[28] V. Sreedhar, M. Burke, and J. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In Proc. *PLDI,* Van Couver, Canada, 2000.

[29] J. M. Stichnoth, G-Y. Lueh, and M. Cierniak. Support for garbage collection at every instruction in a java compiler. In Proc. *Programming Language Design and Implementation,* Atlanta, 1999.

[30] C.-L. Su and A. M. Despain. Cache design trade-offs for power and performance optimization: A case study, In Proc. *International Symposium on Low Power Electronics and Design,* pp. 63–68, 1995.

[31] *The Sun Labs Virtual Machine for Research.* http://www.sun.com/research/java-topics

[32] Ali-Reza Adl-Tabatabai et al. Fast, effective code generation in a Just-in-Time Java compiler. In Proc. *PLDI,* 1998.

[33] Memories are Forever, *Technology Review,* May-June 2000, pp. 28.

[34] M. C. Toburen. Power Analysis and Instruction Scheduling for Reduced di/dt in the Execution Core of High-Performance Microprocessors, *Master's Thesis,* Dept. of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, May 1999.

[35] http://www.transmeta.com/articles/

[36] N. Vijaykrishnan. *Issues in the Design of a Java Processor Architecture.* Ph.D. thesis, College of Engineering, University of South Florida, 1998.

[37] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In Proc. *the International Symposium on Computer Architecture,* Vancouver, British Columbia, June 2000.

[38] B-S. Yang et. al. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In Proc. *PACT'99,* California, October 1999.

[39] W. Ye. *Architectural Level Power Estimation and Experimentation.* Ph.D. Thesis, Department of Computer Science and Engineering, The Pennsylvania State University, October 1999.

On the Software Virtual Machine for the Real Hardware Stack Machine

Takashi Aoki
Autonomous System Laboratory,
Fujitsu Laboratories Limited

Takeshi Eto
Semiconductor Group,
Fujitsu Limited

Abstract

Several technologies for Java[1] [1] program execution have been reported, e.g., Just-In-Time (JIT) compilation, pre-compilation engine, etc., to improve its running speed. Bytecode engine is another approach by taking advantage of the hardware acceleration.

This paper is concerned with the brief introduction to the picoJava-II core technology and its implementation at Fujitsu. Then, we will present our software mapping approach onto the hardware stack machine, focusing on Fujitsu's picoJava-II implementation, MB86799 [2]. Finally, we will report some benchmark results of Java virtual machine execution on the real stack machine and discuss yet unresolved problems.

1 Introduction

Our work aims at combining hardware bytecode stack machine with software Java virtual machine. This section explains how picoJava-II architecture works to help you understand our software development and porting work.

[1] Java, picoJava, JDK, J2ME, and PersonalJava are trademark or registered trademark of Sun Microsystems, Inc.

[2] MB86799 is an evaluation chip. Note that our project was not targeted on the specific end-user product, such as mobile phones or PDA's

1.1 picoJava-II Architecture

Sun's picoJava-II core is a Java bytecode execution engine. First of all, we briefly describe the architecture of picoJava-II core for the better understanding of our software approach.

McGhan et al. [2] reported the core technology of the picoJava architecture. Sun published the Programmer's Reference Manual[3]. Data sheets and other technical documentation are available through Sun Community Source Licensing (SCSL) program[4] as well.

1.1.1 Registers

The picoJava-II architecture has registers for specific purposes as depicted in Table 1. Among these, it is notable that the stack registers, the constant pool register, the monitor caching registers, and the garbage collection register are designed to contribute to the performance improvement of Java program execution. Their access can be realized by simple register read/write operation in picoJava-II. On the other hand, these registers are assigned to the variables in the software virtual machine implementation, such as JDK, instead.

Table 1: picoJava-II Registers

Category	Register
Program Counter	PC
Stack Registers	VARS FRAME OPTOP OPLIM SC_BOTTOM
Constant Pool	CONST_POOL
Memory Protection	USERRANGE1 USERRANGE2
Program Status	PSR
Monitor Caching	LOCKCOUNT0 LOCKCOUNT1 LOCKADDR0 LOCKADDR1
Trap	TRAPBASE
Garbage Collection	GC_CONFIG
Breakpoint	BRK1A BRK2A BRK12C
Global	GLOBAL0 GLOBAL1 GLOBAL2 GLOBAL3

1.1.2 Instructions

The picoJava-II supports 226 bytecode instructions which Java Virtual Machine Specification [5] defines and 115 extended instructions added in picoJava-II Programmer's Reference Manual. The extended instructions are capable of either one of the following functions.

- Direct memory access
- C language interface
- Cache manipulation

1.1.3 Traps

Trap handlers can be divided into the following categories.

- Instruction emulation
- Hardware origin
- Monitor cache interface
- Garbage collection interface

Some of the Java bytecode instructions are too complicated to implement in hardware, such as *new*. This is originated in the fact that these instructions are strongly related to the constant pool resolution. Therefore, the system programmers should write the emulation code for such instructions for picoJava-II and the specific Java platform.

Similar to the conventional microprocessors, the picoJava-II core notifies the error condition in hardware as a trap of the software. The errors include wrong memory alignment, illegal instruction, and so on.

As we see above, the picoJava-II core supports not only the monitor cache register but also its bytecode instructions, *monitorenter* and *monitorexit* in hardware. When the exception occurs in the monitor cache interface of hardware, a trap is notified to the software. Then, it is up to the software's task to continue the appropriate action for the monitor.

The picoJava-II core has a register for the garbage collector. When the garbage collection timing can be observed by the hardware, this is also notified to the software.

1.2 The picoJava-II core features

The picoJava-II core directly executes Java bytecode instructions defined in picoJava-II Programmer's Reference Manual and supports extended bytecode instructions which enable a direct memory access. Its instruction set also includes the *_quick* instructions by which the resolved object can be handled in a much faster manner. The instructions that require constant pool resolution are processed in the trap handler software followed by the trap. The picoJava-II core has the following features to improve the Java application performance.

- Stack cache

- Stack dribble
- Instruction folding
- Write barrier support
- C function interface

1.2.1 Stack Cache

As the Java VM Specification defines a stack machine architecture, the picoJava-II core is based on a stack machine architecture. Software implementation JVM, such as JDK, uses a straightforward memory area as the Java stack. On the other hand, the picoJava-II core takes advantage of the hardware cache for the stack. This improves the bytecode execution performance. The Java stack cache of the picoJava-II core consists of 64 word entries. Once the stack pointer grows/shrinks beyond hardware limits, stack dribbling starts and the contents of the cache are moved to and from the memory. Therefore, the stack cache is transparent to the software.

1.2.2 Stack Dribble

The picoJava-II caches the top 64 entries of the stack. When the software accesses the stack within this range, the core takes advantage of benefits from the high performance of the stack access. Here, one of the following two operations will be started by the hardware. The *spill* writes entries in the stack cache out to the data cache to make some space for new value, while the *fill* reads entries into the stack cache from the data cache to provide the value for the coming instruction. These are operated concurrently in the background.

1.2.3 Instruction folding

The picoJava-II core provides a solution to the access inefficiency problem of stack machines. Its stack cache is actually a full random access register file, so the pipeline is capable of the immediate access of all 64 entries in the stack cache.

```
iload_1    add local1,local2,local3
iload_2
iadd
istore_3
```
(a) (b)

Figure 1: Instructions to Add Two Locals

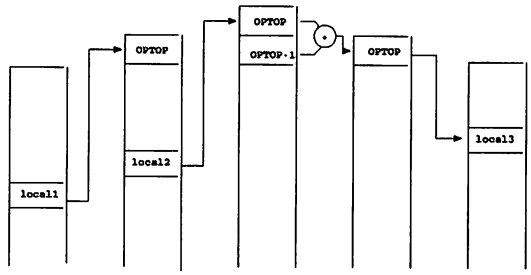


Figure 2: Execution without Instruction Folding

The execution technique called *instruction folding* takes advantage of this. When two values are added in the stack machine, the instruction sequence depicted in Figure 1 (a) is used. This is also visualized in Figure 2.

However, once the hardware detects the two entries are in the stack register file, it can add the values and store the result to the stack cache in one sequence with the instruction folding technique as seen in Figure 1 (b) and Figure 3.

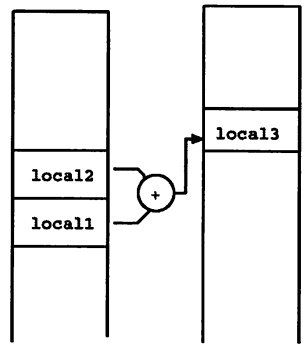


Figure 3: Instruction Folding

1.2.4 Write Barrier Support

The picoJava-II core has the garbage collection accelerating facility called *write-barrier* register. When the accessed object resides in the mem-

ory area beyond the write-barrier, the picoJava-II core detects it and notifies the software of the event by *gc_notify* trap. The garbage collection condition can be controlled by `GC_CONFIG` register. Utilizing this register is strongly related to the garbage collection algorithm ([8] and [9]).

1.2.5 C Function Interface

While the Java method is invoked by the method invoke instructions, such as *invokevirtual* or *invokespecial*, as Java Virtual Machine Specification defines, C language functions invocations relies on the platform dependent implementation. The *call* instruction is added to the picoJava-II core to support the C *native* function. This also accelerates the Java Native Interface (JNI) method invocation as well as the internal JVM functions written in C.

1.3 Fujitsu MB86799

Fujitsu's implementation of picoJava-II technology, MB86799 [3] , consists of picoJava-II core, the external bus interface, and PCI bus interface. This implementation has an 8KB instruction cache, an 8KB data cache, a 64-entry stack cache, and a floating point unit as shown in Figure 4. The MB86799 can run at the external maximum frequency rate 33MHz and internal 66MHz. The frequency ratio between the internal and the external can be varied from 2 to 5. The chip consumes 360mW with the source 2.5V at 66MHz.

2 Overview of the Software

We now present a high-level overview of the software implementation on our picoJava-II chip.

We ported Sun's PersonalJava 3.02 to picoJava-II running Fujitsu's real-time OS, REALOS[4], a variant of μ-ITRON RTOS. Figure 5 is an overview of our system software.

[3]MB86799 is based on β-version specification of the picoJava-II core. There is a minor difference of the instruction operation between the β version and the current FCS release.

[4]REALOS is a trademark of Fujitsu Limited.

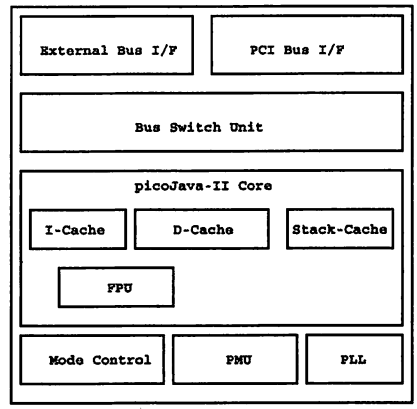


Figure 4: MB86799 Block Diagram

We did not write our own software virtual machine for picoJava-II from scratch, instead we ported PersonalJava because of the following reasons.

- Porting was estimated to be finished in a shorter time.
- Scratch-built virtual machine would be more difficult to accomplish the compatibility with reference software.

The source code of the latest PersonalJava, namely PersonalJava 3.1, can be obtained through SCSL[6].

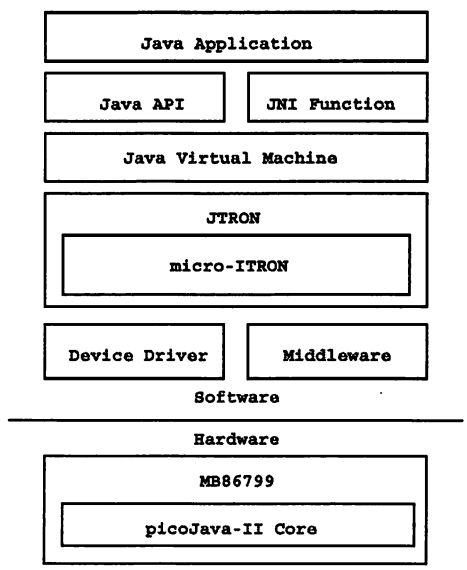


Figure 5: Overview of Software Structure

2.1 Operating System

We were motivated to use the REALOS operating system in order to meet the strong demand for real-time assurance support from the embedded device community. REALOS is μ-ITRON RTOS specification compliant. All the operating facilities are provided by REALOS and our JVM takes advantage of them. The μ-ITRON specification and its Java interface variant, JTRON, specification are available at TRON Project Home Page[7].

2.2 Java Virtual Machine

Now that we are able to execute the bytecode on hardware directly, the term *virtual machine* may be somewhat incorrect. However, since no appropriate naming has been proposed so far, let us continue to call it *virtual machine* at the moment. As mentioned above, our virtual machine is based on PersonalJava 3.02. The five items we had to consider while porting the virtual machine to the picoJava-II chip are as follows.

1. Object data structures
2. Exception handling
3. JNI support
4. Lock register
5. JCC (JavaCodeCompact)

3 Porting Strategy

We show here our strategy to port PersonalJava onto picoJava-II architecture, focusing on how to adapt software virtual machine for direct bytecode execution engine. We use C and assembler language to implement the virtual machine.

3.1 Object Data Structure

The picoJava-II core directly executes the Java method and handles the Java object. This implies objects must be placed on memory in the exact format as the core expects. Since PersonalJava is derived from JDK, its object format is different from the one that picoJava-II specifies. Figure 6 and 7 are the examples of the fixed object format for PersonalJava and picoJava-II respectively. In picoJava-II, the array reference always points to the array header position, which is followed by an array length. The actual array data follows after the length field. The array contents can be accessed by a reference and indexing.

We modified the array structure definitions and their access macro definitions of PersonalJava so that picoJava-II hardware can manipulate the array structure directly.

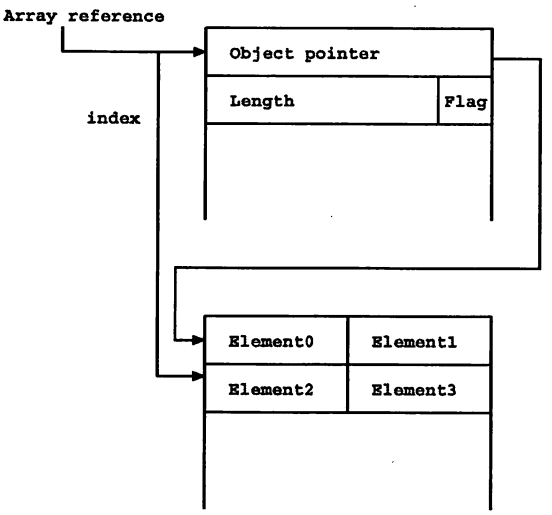


Figure 6: An Example of PersonalJava Array - short primitive

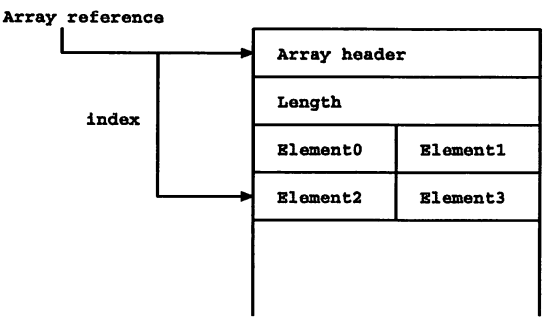


Figure 7: An Example of picoJava-II Array - short primitive

3.2 Exception Handling

We now describe the following four topics on the difficult handling of the exception for picoJava-II.

1. Stack Usage
2. Stack Cache Coherency
3. Stack Growing Direction
4. Exception Origin

3.2.1 Stack Usage

We noticed the difference of stack usage between the JDK/PersonalJava JVM implementation and the picoJava-II handling. While the former uses the two distinct stack, i.e., C stack and Java stack, in its memory area, the latter uses only one stack which is shared among JVM internal C functions, the JNI method, and the Java method. In the former case, the JVM simply traces the frame structure in the Java stack to find an appropriate exception handling method when exception occurs. However, in picoJava-II, it is not that straightforward. The picoJava-II core defines three kinds of frame format, Java method frame, trap handler frame and C function frame. When an exception occurs, the exception handler search routine tries to locate the Java method frame that catches it, skipping trap handler frames and C frames. Both frames are not a concern for Java exception.

In addition, in our approach, the JNI method is implemented by the Java stub method as described below. Therefore, we have to ignore the Java stub method frame for the JNI call as well.

Figure 8 is an example of the exception tracing. Here, the method with the frame (2) invokes another method (1) via a trap frame. Each frame is linked as (a) and (b). As explained above, the trace method routine needs to behave as if the method frames were linked as (c) in order to trace them. JNI case can be explained by substituting the trap frame by a JNI stub frame.

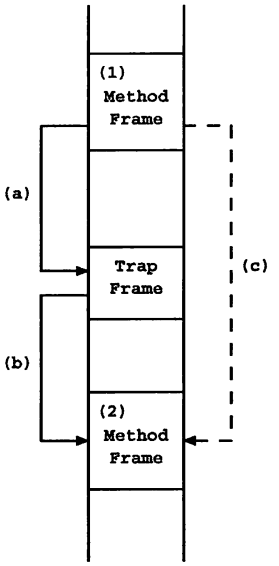


Figure 8: Exception Table Trace

3.2.2 Stack Cache Coherency

There is another difficulty for traversing method frames on the picoJava-II stack. When we scan the stack, we usually use a pointer addressing the stack. To do that on picoJava-II core, we have to flush the stack cache before accessing the stack frame, since there is no coherency between its stack cache and the data cache.

But flushing the stack cache is a time consuming operation. By using *load/store* instructions [5], one can access the stack data correctly without a stack cache flush. To traverse method frames, however, we must set the **VARS** register to an appropriate address successively. This is also cumbersome, as we can't use local variables while changing the **VARS** register.

We used the stack cache flush scheme for PersonalJava 3.02 port, as it was simpler.

3.2.3 Stack Growing Direction

The stack growing direction also prevents us from manipulating frame data directly. The picoJava-II

[5] Local variables are accessed by the offset index from **VARS** with *load/store* instructions, such as *iload_0* and *astore_1*. **VARS** is a register that points to the first argument address in its Java stack or C stack.

Java stack grows downward, i.e., from the higher address to the lower. Figure 9 shows Java frame format (a) and trap frame format (b) in picoJava-II stack. Note that **FRAME** register always points to the previous or return **PC** address and the position is located at the highest offset or the middle of the data.

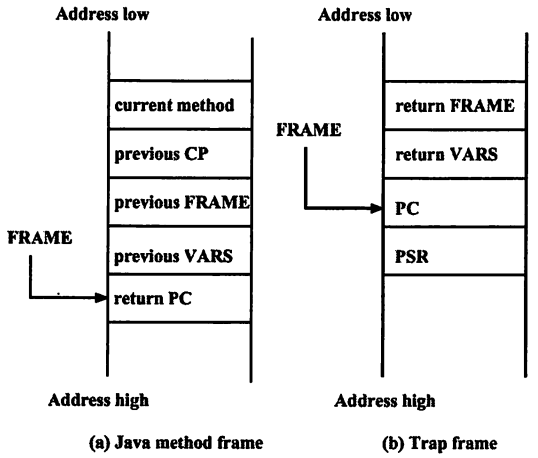


Figure 9: Stack Growing Direction

```
struct javaFrame {
   METHOD          *currentMethod;
   CONSTPOOL       *previousCONSTPOOL;
   FRAME           *previousFRAME;
   VARS            *previousVARS;
   unsigned char *returnPC;
};
```

Figure 10: Java Frame in C Structure

On the other hand, when we refer the stack frame by a data structure in C language, a pointer for the structure always points at the lowest address. Thus, when we trace the Java frame data in the high level language such as C, we have to declare the frame structure in the reverse order from the picoJava-II core definition, as seen in Figure 10. Then we have to recalculate the next frame's start address using `previousFRAME` field value every time we move the pointer.

3.2.4 Exception Origin

Some of the exceptions, e.g., NullPointerException, can be raised by both software and hardware in picoJava-II. When the instructions detect the null reference, the hardware automatically raises the exception trap. This has initially been the responsibility of the virtual machine software, and it still exists on picoJava-II. One has to take into account both of the exception trap case and the software originated exception to save the code space.

3.3 JNI

Again the term JNI is not an exact term since the native instruction set of picoJava-II is the bytecode. But let us call the interface between a Java method and a function written in C, JNI here.

As shown in Figure 11, the calling convention of Java method is different from that of C function on picoJava-II. Re-pushing the argument variable to the stack is necessary when the C function is invoked from a Java method. The method is invoked by invoke instructions, for example, *invokevirtual_quick*, on picoJava-II, whether it is written in C or Java. In other words, the method is invoked in a different manner depending on its access flag as defined in Java Virtual Machine Specification when implemented by software. However, the hardware picoJava-II does not care about such a flag but always invokes the method as if it were written in Java.

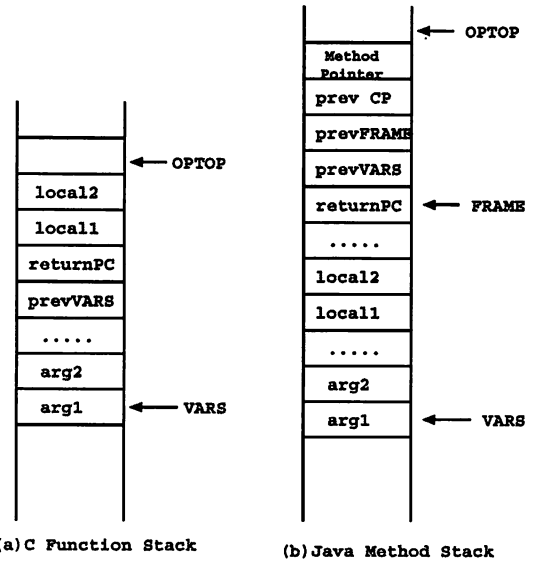


Figure 11: C and Java Stack Usage

There are two approaches to resolve this problem as explained in picoJava-II Reference Manual .

- Create a stub method between the calling Java method and the callee C function
- Invoke the C function from the trap software code of *invoke* instructions

We chose the former strategy since apparently the latter sacrifices the performance without using the *_quick* instruction of hardware.

3.4 Lock Register

Java virtual machine controls the shared object data with a monitor by locking with mutex. The picoJava-II core has two pairs of **LOCKCOUNT** and **LOCKADDR** registers for this purpose. The *monitorenter* instruction has the following logic:

- Compare the object address with the value of either one of **LOCKADDR** registers.
- If the address matches, increment the corresponding **LOCKCOUNT** register.
- Otherwise, the trap handler is invoked.

This indicates that when the object is not actually shared, the mutex is accomplished merely by setting the object address to the **LOCKADDR** register and incrementing or decrementing the contents of **LOCKCOUNT** register. As a result, the overhead for the monitor creation is significantly reduced.

The virtual machine also needs to manipulate the *synchronized* method code in order to utilize the hardware *monitorenter* and *monitorexit* instructions as follows.

1. Replace all *return*, *areturn*, *ireturn*, *lreturn*, and *dreturn* instructions with the extended *exit_sync_method* instruction. [6]
2. Insert code set shown in Table 2 and 3, depending on the attribute of the method.
3. Change the exception table for the method so that `start_pc`, `end_pc`, and `handler_pc` of the entries are each incremented by 8 or 12 to map onto the relocated code.

[6] All the bytecodes in the method must be scanned in order to locate and replace the *X*return instruction due to the variable length of the bytecode instruction set. Analysis involving *lookupswitch*, *tableswitch*, and *wide* is often complicated.

Table 2: Code Prepended to Synchronized Non-static Methods

Address	Instruction	Action
0	*aload_0*	Get the object reference
1	*monitorenter*	Synchronize on the object reference
2	*jsr*+6	Push PC on top of the stack, which is also **FRAME** - 20, and jump to the next instruction.
5	*aload_0*	Get the object reference
6	*monitorexit*	Exit the monitor.
7	*Xreturn*	Return to the caller of the correct type.

Table 3: Code Prepended to Synchronized Static Methods

Address	Instruction	Action
0	*get_current_class*	Get the current class pointer.
2	*monitorenter*	Synchronize on the class pointer.
3	*jsr*+9	Push PC on top of the stack, which is also **FRAME** - 20, and jump to the original code.
6	*get_current_class*	Get the current class pointer.
8	*monitorexit*	Exit the monitor.
9	*Xreturn*	Return to the caller of the current type.
10	*nop*	
11	*nop*	Ensure that the code is a multiple of 4 bytes to prevent changes in padding for lookupswitch and tableswitch.

In our implementation, the code is manipulated in the class loader. However, if the system developer is aware that the synchronized methods are rarely called, the manipulation can be done at the time of the method invocation, such as in the trap handler.

3.5 JavaCodeCompact (JCC)

PersonalJava has a tool called JavaCodeCompact (JCC) to improve the class loading performance and reduce the code size. The tool converts the class file into the runtime memory image and links it with the virtual machine. Since the picoJava-II has the unalterable object format which the hardware accepts, and the format differs from that of the software implementation, we had to modify the internal data structure and code generation part of the tool accordingly as well.

4 Benchmark Results

Table 4 shows the Embedded CaffeineMark benchmark result of MB86799. The table contains the actual results. PJEE is the emulation environment of PersonalJava, which has an interpreter loop written in C. We used JDK1.1.8 to compare with a JIT compiler result. But please note that the implementation, especially the data structure, is different from PJEE. The number indicates that the larger it is, the faster the platform is.

Table 4: Embedded CaffeineMark Results

	MB86799 66MHz /33MHz PJ 3.02	Pentium-III 700MHz /100MHz PJEE 3.02	Pentium-III 700MHz /100MHz JDK 1.1.8
Sieve	395	367	11644
Loop	984	339	40248
Logic	667	336	168150
String	594	957	18867
Float	407	333	18981
Method	593	362	18116

the larger the faster

Table 5 is the results of the Java version of Tak function[11]. The Tak is a heavy recursion test program to evaluate function call performance. Our experiment was undertaken with the function call of the argument Tak(18,12,6) for 1000 times.

Table 6 shows the results of Java vesion of Linpack [12] benchmark program.

Table 7 is the result comparison of SPECjvm98 [13] benchmark programs. The number here indicates that the smaller, the faster the platform is. Note that some of the SPECjvm98 programs are not listed here since they are too large to run on our embedded system. Please note that Pentium-III runs more than ten times faster than picoJava-II for internal clock and three times faster for external bus clock.

Table 5: Tak Benchmark Results

MB86799 66MHz /33MHz PJ 3.02	Pentium-III 700MHz /100MHz PJEE 3.02	Pentium-III 700MHz /100MHz JDK 1.1.8
24392msec	38670msec	1593msec

the smaller the faster

Table 6: Linpack Results

MB86799 66MHz /33MHz PJ 3.02	Pentium-III 700MHz /100MHz PJEE 3.02	Pentium-III 700MHz /100MHz JDK 1.1.8
1.798Mflops/s 0.38sec	1.761Mflops/s 0.39sec	22.889Mflops/s 0.03sec

Table 7: SPECjvm98 Results

	MB86799 66MHz /33MHz PJ 3.02	Pentium-III 700MHz /100MHz PJEE 3.02	Pentium-III 700MHz /100MHz JDK 1.1.8
jess	1109.189	164.597	51.674
mpegaudio	400.398	610.888	14.661
mtrt	786.019	174.260	30.564
jack	1534.797	204.043	31.295

the smaller the faster

Among the benchmark results here, the Java version of Linpack runs almost as fast on picoJava-II as on JDK JIT, if the number is normalized by internal clock. Figure 12 is the bytecode statistics during its execution. The bytecode instructions here are categorized as follows:

- Stack manipulation
  iload,aload,istore,astore,push, pop,dup,inc, iconst_0, etc
- Arithmetic implemented in hardware
  iadd, fmul, dmul, etc

- Array load/store
  caload, *bastore*, etc
- Conditional branch
  ifeq, *if_acmpeq*, etc
- Unconditional branch
  goto, *ireturn*, etc
- Field access
  getstatic, *putfield*, etc

All of these categories are implemented in hardware on picoJava-II, and consists of 96.7% of the instructions executed. Obviously, this is very advantageous in achieving good Java application performance on picoJava-II. Figure 13 is the bytecode composition of Linpack class file. 93.2% instructions of the class file are implemented directly in hardware on picoJava-II. The native methods, which often become the bottle neck for picoJava-II execution performance, invoked by Linpack are java.lang.System.currentTimeMillis() and some string manipulation method to produce the result message. The fact implies Java program performance for picoJava-II can be evaluated statically in advance by analizing the instruction category.

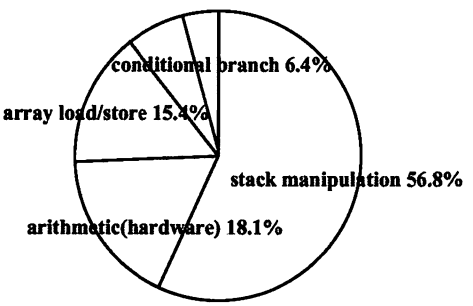


Figure 12: Linpack bytecode execution statistics

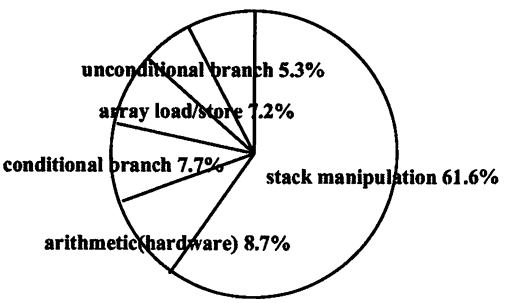


Figure 13: Linpack bytecode composition

The ratio of the stack manipulation instructions of Embedded CaffeineMark is as much as that of Linpack, as seen in Figure 14. However, when the hardware implemented field access instruction, such as *getfield_quick*, is operated, its corresponding emulated instruction, *getfield*, for example, is executed beforehand. In addition to this, a large amount of time is spent on the string manipulation native methods written in C for this benchmark test. This is fairly disadvantageous for picoJava-II technology.

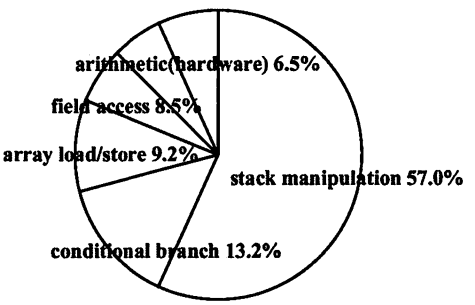


Figure 14: Embedded CaffeineMark bytecode execution statistics

Our benchmark results indicate that our approach on the real hardware stack machine overwhelms the performance of the corresponding C interpreter. In addition, the direct execution of bytecode compares competitively with JIT enabled virtual machine for the bytecode oriented Java applications.

The Java microprocessor technology can be an adequate solution to the strong demand from the embedded Java community.

5 Future Work

As reported by Gu et al. [10], there are a number of JVM code tuning techniques. These can be also applied to our virtual machine since ours is derived from Sun's JDK implementation. For instance, the String result of Embedded CaffeineMark benchmark was improved by about 10% with a few function inlining within EE() function in our experiment. This is still experimental in our work and we continue the code tuning focused on the specific implementation.

We have not used the power of the picoJava-II technology to its full extent yet, especially in its garbage collection support area. We will continue to improve the performance of PersonalJava platform. Besides this, the current PersonalJava is based on JDK 1.1.x technology. However, the present embedded Java community request varies in wide range. Porting the J2ME platform will be one of our main concerns in the near future.

6 Open Problems

There have still been open problems found in our experience. In this section, we report them for the future hardware design and its tool improvement.

6.1 Stack Cache

The current picoJava-II technology heavily depends on the cache access for its performance improvement. However, because of the difficulty of predicting the cache residency, the software performance sometimes varies with a minor modification. This often makes system performance tuning difficult. Besides, the lack of coherency between the stack cache and the data cache often complicates the software designing, especially for those that access the local stack directly, such as trap handlers for the bytecode instruction emulation, for example *invokeinterface* or *getfield*.

6.2 C Language Interface

At the time of writing of this paper, the only C compiler which supports the picoJava-II extended instruction set is MetaWare's High C/C++[7]. The compiler generates the picoJava-II C function interface object code in a straightforward manner. In other words, while the compiler generates the function code with *call/return* by register interface, the Java class method expects the code with *invoke/return* by stack interface.

Figure 15 depicts an example of this problem. Initially, Java method (3) invokes the C function (1) as a native method. Although the object code of the function (1) returns the value by setting it to register `GLOBAL1`, the caller method (3) expects the value in the Java stack. In our approach, we insert the stub method (2) between them.

One would notice that there are a number of C native methods in the Java system class. For example, java.lang.Math.pow() function only returns the native pow() function result. As we explained earlier, we have two options to support JNI on picoJava-II, creating a stub method or call the C function in the trap handler without using the *_quick* hardware instructions. If the C compiler could compile a C function following the Java method calling interface, the JNI performance would be improved significantly.

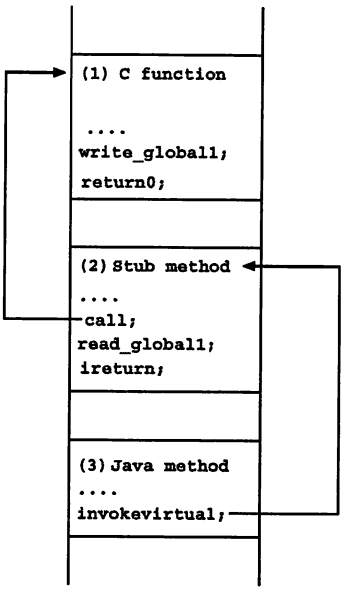


Figure 15: C-Java Language Interface Problem

6.3 Aggregate Stack Problem

The picoJava-II core provides a stack named *aggregate stack*[8] for the C language local variables whose addresses are used like arguments passed by reference. Aggregate stack is designed to solve the stack cache incoherency problem. However, this is sometimes inclined to increase the overhead in the JNI functions.

The stack often complicates the system program-

[7] High C/C++ is a trademark of MetaWare, Inc.

[8] The aggregate stack is a portion of memory area pointed by `GLOBAL0` register.

mer's work because it is difficult to handle the aggregate stack both by the C compiler and the hand-written assembler code. Also, system programmers are required to be careful that the aggregate stack must be released appropriately when an exception occurs in a Java method and the exception is caught beyond a JNI function that uses the aggregate stack.

7 Conclusion

In this paper, we have shown our porting strategy of the software Java virtual machine onto the real hardware stack machine using picoJava-II microprocessor as an example.

The direct bytecode execution engine demonstrates that it can be even competitive with JIT enabled software virtual machine, especially for the bytecode oriented applications. Our benchmark results also indicate that Java microprocessor can be one of the effective solutions for the embedded Java technology where low power consumption, small memory footprint and quick start are preferred.

The picoJava-II core is a well defined CPU to run not only Java but also C, although there still exists some room for improvement. Some will be solved by hardware modification and others by improving the C compiler.

References

[1] Gosling, J., Joy, B., Steele, G., The Java Language Specification, Addison Wesley, 1996

[2] McGhan, H., O'Connor, M., PicoJava: A Direct Execution Engine For Java Bytecode. In IEEE Computer Vol 31, No 10, October 1998

[3] Sun Microsystems, Inc., picoJava-II Programmer's Reference Manual, March 1999

[4] Sun Microelectronics, picoJava Microprocessor Cores, *http://www.sun.com/microelectronics/picoJava/*

[5] Lindholm, T., Yellin, F., The Java Virtual Machine Specification, Addison Wesley, 1997

[6] Sun Community Source Licensing, Source Code Catalog, *http://www.sun.com/software/communitysource/*

[7] The ITRON Project, JTRON2.0 Specification, *http://tron.is.s.u-tokyo.ac.jp/TRON/ITRON/home-e.html*

[8] Grarup, S., Seligmann, J., Incremental Mature Garbage Collection, M.Sc. Thesis, Aarhus University, Computer Science Department, August 1993.

[9] Hudson, R., Moss, J.E.B., Incremental Garbage Collection For Mature Objects, Proceedings of International Workshop on Memory Management, St. Malo, France, September 16-18, 1992

[10] Gu, W., Burns, N.A., Collins, M.T., Wong, W.Y.P., The Evolution of a high-performing Java virtual machine, IBM Systems Journal, Vol. 39, No. 1, 2000, IBM Corporation

[11] Gabriel, R.D., Performance and Evaluation of Lisp Systems, The MIT Press, 1985

[12] Linpack Benchmark - Java Version, Netlib Repository, *http://www.netlib.org/benchmark/linpackjava/*

[13] SPEC JVM98 Benchmarks, *http://www.spec.org/osg/jvm98/*

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

Supporting Members of the USENIX Association:

Addison-Wesley
Kit Cosper
Earthlink Network
Edgix
Interhack Corporation
Interliant
Lessing & Partner
Linux Security, Inc.
Lucent Technologies
Microsoft Research
Motorola Australia Software Centre
New Riders Press
Nimrod AS
O'Reilly & Associates Inc.
Raytheon Company
Sams Publishing
Sendmail, Inc.
Smart Storage, Inc.
Sun Microsystems, Inc.
Sybase, Inc.
Syntax, Inc.
Taos: The Sys Admin Company
UUNET Technologies, Inc.

Supporting Members of SAGE:

Certainty Solutions
Collective Technologies
Electric Lightwave, Inc.
ESM Services, Inc.
Lessing & Partner
Linux Security, Inc.
Mentor Graphics Corp.
Microsoft Research
Motorola Australia Software Centre
New Riders Press
O'Reilly & Associates Inc.
Raytheon Company
Remedy Corporation
RIPE NCC
Sams Publishing
SysAdmin Magazine
Taos: The Sys Admin Company
Unix Guru Universe

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-880446-11-1